VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

DIPLOMOVÁ PRÁCE

Brno, 2018 Bc. Pavel Röszler

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ

DEPARTMENT OF TELECOMMUNICATIONS

IMPLEMENTACE ALGORITMU DEKOMPOZICE MATICE A PSEUDOINVERZE NA FPGA

IMPLEMENTATION OF MATRIX DECOMPOSITION AND PSEUDOINVERSION ON FPGA

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. Pavel Röszler**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. David Smékal**
SUPERVISOR

BRNO 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní obor **Telekomunikační a informační technika**
Ústav telekomunikací

Student: Bc. Pavel Röszler ***ID:*** 164384
Ročník: 2 ***Akademický rok:*** 2017/18

NÁZEV TÉMATU:

Implementace algoritmu dekompozice matice a pseudoinverze na FPGA

POKYNY PRO VYPRACOVÁNÍ:

Práce se zabývá dekompozicí hermitovských, pozitivně-semidefinitních komplexních čtvercových matic a pseudoinverzní obecně obdelníkové komplexní matice. Zaměřte se na výpočet vlastních čísel a vlastních vektorů (příp. singulárních) a uvažujte také metody umožňující výpočet pouze vlastního vektoru příslušejícího dominantnímu vlastnímu číslu. Maximální velikost matice uvažujte m=4. Na vývojovém kitu implementujte a optimalizujte vybrané metody vhodné pro implementaci na platformu FPGA a ověřte jejich funkčnost. Posuďte závislosti výpočetní náročnosti na velikosti matice a zhodnoťte přesnosti výpočtu. V závěru diskutujte efektivitu výpočtu a závislost výpočetní náročnosti na velikosti matice.

DOPORUČENÁ LITERATURA:

[1] QUARTERONI, Alfio, SACCO, Riccardo, SALERI, Fausto. Numerical Mathematics. 1. vyd., Berlín, 2007, 655 s. ISBN 0939-2475.

[2] WOLF, Wayne. FPGA based system design. New Jersey: Prentice-Hall, 2004, 530 s. ISBN 0-13-142461-0.

Termín zadání: 5.2.2018 ***Termín odevzdání:*** 21.5.2018

Vedoucí práce: Ing. David Smékal
Konzultant: Ing. Antonín Heřmánek, Ph.D., ERA a.s.

prof. Ing. Jiří Mišurec, CSc.
předseda oborové rady

ABSTRAKT

Cílem této práce je implementace výpočtů vlastních čísel a vektorů a výpočet pseudoinverze matice na hradlovém poly. Při těchto výpočtech se velmi často používají maticové rozklady, které jsou popsány jako první. Následuje shrnutí teorie a uvedení jednotlivých metod, z nichž některé byli implementovány v *Matlab*. Pro implementaci do FPGA (*Field Programmable Gate Array*) je využito nástrojů a knihoven *Vivado High-Level Synthesis*, v práci je stručný popis problematiky FPGA obvodů a jejich programování a detailní popis principů a možností nástrojů HLS s důrazem na funkce z knihovny pro lineární algebru, které jsou následně využity v jednotlivých variantách výpočetních bloků. Výsledky jednotlivých variant jsou dále srovnány z hlediska časování a využití prostředků FPGA. Vybraný blok byl ověřen na vývojovém kitu a analyzována jeho numerická přesnost na základě dat z měření.

KLÍČOVÁ SLOVA

Vlastní čísla, vlastní vektory, pseudoinverze, dekompozice, QR, ortogonalizace, singulární čísla, singulární vektory, SVD, High-Level Synthesis, FPGA.

ABSTRACT

The purpose of this thesis is to implement algorithms of matrix eigendecomposition and pseudoinverse computation on a Field Programmable Gate Array (FPGA) platform. Firstly, there are described matrix decomposition methods that are broadly used in mentioned algorithms. Next section is focused on the basic theory and methods of computation eigenvalues and eigenvectors as well as matrix pseudoinverse. Several examples of implementation using *Matlab* are attached. The *Vivado High-Level Synthesis* tools and libraries were used for final implementation. After the brief introduction into the FPGA fundamentals the thesis continues with a description of implemented blocks. The results of each variant were compared in terms of timing and FPGA utilization. The selected block has been validated on the development board and its arithmetic precision was analyzed.

KEYWORDS

Eigenvalues, eigenvectors, pseudoinverse, decomposition, eigendecomposition, QR, orthogonalization, singular values, singular vectors, SVD, High-Level Synthesis, FPGA.

RÖSZLER, Pavel. *Implementace algoritmu dekompozice matice a pseudoinverze na FPGA*. Brno, 2018, 77 s. Diplomová práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací. Vedoucí práce: Ing. David Smékal

PROHLÁŠENÍ

Prohlašuji, že svou diplomovou práci na téma „Implementace algoritmu dekompozice matice a pseudoinverze na FPGA“ jsem vypracoval(a) samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor(ka) uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil(a) autorská práva třetích osob, zejména jsem nezasáhl(a) nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom(a) následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

podpis autora(-ky)

PODĚKOVÁNÍ

Rád bych poděkoval svým kolegům ze společnosti ERA a.s.. Děkuji panu Ing. Antonínovi Heřmánkovi, Ph.D. ze za odborné vedení, konzultace a podnětné návrhy k práci. Děkuji Adamovi Koutnému za konzultace, připomínky a přípravu testovacích dat z měření. Poděkování patří také vedoucímu diplomové práce Ing. Davidovi Smékalovi za důležité rady a připomínky v průběhu vypracování práce. Za podporu při studiu bych hrozně rád poděkoval svoji rodině a přátelům.

Brno ....

podpis autora(-ky)

Faculty of Electrical Engineering
and Communication
Brno University of Technology
Purkynova 118, CZ-61200 Brno
Czech Republic
http://www.six.feec.vutbr.cz

PODĚKOVÁNÍ

Výzkum popsaný v této diplomové práci byl realizován v laboratořích podpořených z projektu SIX; registrační číslo CZ.1.05/2.1.00/03.0072, operační program Výzkum a vývoj pro inovace.

Brno ....

podpis autora(-ky)

EVROPSKÁ UNIE
EVROPSKÝ FOND PRO REGIONÁLNÍ ROZVOJ
INVESTICE DO VAŠÍ BUDOUCNOSTI

OBSAH

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

ÚVOD

Tato práce se zabývá implementací metod pro výpočet vlastních čísel a vektorů a Moore-Penrosovy pseudoinverze na platformě FPGA.

V první části se práce věnuje rozkladům matice, kde je velká pozornost věnována především QR rozkladu, protože je základem pro výpočet vlastních čísel a vektorů pomocí QR algoritmu. Dále je možné využít QR rozklad pro určení pseudoinverze pomocí přímého výpočtu nebo prostřednictvím singulárního rozkladu. Způsobů, jakými se QR rozklad provádí, existuje celá řada. V práci jsou uvedeny a srovnány základní algoritmy, mezi které patří Gram-Schmidtův proces, Householderovy reflexe a Givesovy rotace. Další metody pro výpočet vlastních čísel jsou Jacobiho metoda a metoda Rayleigho podílu, které se dají využít pro symetrické matice. Pro ověření některých metod, byly před vlastní implementací vytvořeny zdrojové soubory pro *Matlab*, které jsou uvedeny v přílohách.

Pro implementaci algoritmů na hradlovém poli jsou využity nástroje *Vivado High-Level Synthesis*. V práci je popsán úvod do problematiky hradlových polí a jejich programování a pozornost je věnována nástrojům *Vivado High-Level Synthesis*. Tyto nástroje umožňují generování IP bloků, na základě zdrojových souborů v programovacím jazyce C++ a poskytují celou řadu funkcí ve svých knihovnách. V práci je uveden popis tohoto procesu, včetně způsobu testování, optimalizace, volbu rozhraní a export bloků pro další využití.

Samostatná kapitola je věnována implementaci bloků pro výpočet vlastních čísel a vektorů a bloků pro výpočet pseudoinverze matice. Jsou zde popsány jednotlivé variant a jejich vlastnosti, včetně srovnaní jednotlivých variant, ověření zvolené varianty na vývojovém kitu a výsledky testování s naměřenými daty.

1 DEKOMPOZICE MATICE

Dekompozice matice je obecně taková operace, která danou matici rozloží na matice, které mají určité vlastnosti, vhodné pro řešení různých problému. Dekompozic existuje celá řada. V následující kapitole jsou popsány ty, které se vyskytují v algoritmech pro výpočet vlastních čísel a vektorů nebo při výpočtu pseudoinverze matice.

1.1 LU dekompozice

Prvním příkladem je LU dekompozice matice $\mathbf{A} \in \mathbb{C}^{n\times n}$

$$\mathbf{A} = \mathbf{LU}, \tag{1.1}$$

kde $\mathbf{L}$ je dolní trojúhelníková matice a $\mathbf{U}$ je horní trojúhelníková matice. Tento rozklad je užitečný v případě řešení soustavy rovnic $\mathbf{Ax} = \mathbf{b}$ pro více různých vektorů $\mathbf{b}$. Zavedením nového vektoru $\mathbf{y}$, kde $\mathbf{Ux} = \mathbf{y}$, a následným řešením rovnice $\mathbf{Ly} = \mathbf{b}$ získáváme řešení $\mathbf{Ax} = \mathbf{b}$. Řešení pro dolní trojúhelníkovou matici je možné získat pomocí dopředné substituce. Výsledek je získán dosazením do $\mathbf{Ux} = \mathbf{y}$ [2].

1.1.1 Choleského dekompozice

Tato dekompozice je speciálním případem LU rozkladu, kde $\mathbf{A} \in \mathbb{C}^{n\times n}$ je hermitovská pozitivně semidefinitní matice a $\mathbf{U} = \mathbf{L}^{\mathrm{H}}$ [2].

1.2 QR dekompozice

V případě QR dekompozice chceme určit matice $\mathbf{Q}$ a $\mathbf{R}$ pro které platí

$$\mathbf{A} = \mathbf{QR}, \tag{1.2}$$

kde $\mathbf{A} \in \mathbb{C}^{n\times n}$. Matice $\mathbf{Q}$ je unitární matice, pro kterou platí

$$\mathbf{Q}^{-1} = \mathbf{Q}^{\mathrm{H}}, \tag{1.3}$$

a $\mathbf{R}$ je horní trojúhelníková matice. V případě $\mathbf{A} \in \mathbb{R}^{n\times n}$ se $\mathbf{Q}$ nazývá ortogonální a platí $\mathbf{Q}^{-1} = \mathbf{Q}^{\mathrm{T}}$ [2]. Způsob, kterým získáváme $\mathbf{Q}$ se často označuje jako ortogonalizace. Následuje popis základních ortogonalizačních algoritmů.

1.2.1 Gram–Schmidtův proces

V každém kroku algoritmu je využita operace projekce vektoru $\mathbf{a}$ na vektor $\mathbf{e}$. Projekce využívá podílu skalárních součinů a můžeme ji zapsat jako

$$\mathrm{proj}_{\mathbf{e}}\mathbf{a} = \frac{\langle \mathbf{e}, \mathbf{a} \rangle}{\langle \mathbf{e}, \mathbf{e} \rangle}\mathbf{e}. \tag{1.4}$$

Zapíšeme matici $\mathbf{A} \in \mathbb{R}^{n\times n}$ pomocí jejich sloupcových vektorů jako $\mathbf{A} = [\mathbf{a}_1, \cdots, \mathbf{a}_n]$. Na začátku zvolíme počáteční vektor $\mathbf{u}_1$, jako jeden ze sloupcových vektorů matice $\mathbf{A}$ a určíme vektory $\mathbf{u}_2 \cdots \mathbf{u}_k$ následujícím postupem:

$$\mathbf{u}_2 = \mathbf{a}_2 - \mathrm{proj}_{\mathbf{u}_1}\mathbf{a}_2, \tag{1.5}$$

$$\mathbf{u}_3 = \mathbf{a}_3 - \mathrm{proj}_{\mathbf{u}_1}\mathbf{a}_3 - \mathrm{proj}_{\mathbf{u}_2}\mathbf{a}_3, \tag{1.6}$$

$$\vdots$$

$$\mathbf{u}_k = \mathbf{a}_k - \sum_{j=1}^{k-1} \mathrm{proj}_{\mathbf{u}_j}\mathbf{a}_k. \tag{1.7}$$

Vektory dále normujeme:

$$\mathbf{e}_1 = \frac{\mathbf{u}_1}{\|\mathbf{u}_1\|}, \tag{1.8}$$

$$\vdots$$

$$\mathbf{e}_k = \frac{\mathbf{u}_k}{\|\mathbf{u}_k\|}. \tag{1.9}$$

Ortogonální matice $\mathbf{Q}$ je pak tvořena vektory

$$\mathbf{Q} = [\mathbf{e}_1, \cdots, \mathbf{e}_n]. \tag{1.10}$$

Matici $\mathbf{R}$ určíme jako

$$\mathbf{R} = \begin{pmatrix} \langle \mathbf{e}_1, \mathbf{a}_1 \rangle & \langle \mathbf{e}_1, \mathbf{a}_2 \rangle & \langle \mathbf{e}_1, \mathbf{a}_3 \rangle & \dots \\ 0 & \langle \mathbf{e}_2, \mathbf{a}_2 \rangle & \langle \mathbf{e}_2, \mathbf{a}_3 \rangle & \dots \\ 0 & 0 & \langle \mathbf{e}_3, \mathbf{a}_3 \rangle & \dots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}. \tag{1.11}$$

Nevýhodou tohoto postupu je v některých případech nedostatečná numerická stabilita, proto je často volen alternativní postup popsaný dále.

modifikovaný Gram–Schmidtův proces

Takto upravený proces poskytuje mnohem přesnější numerické výsledky při výpočtu s konečnou přesností. V případě $\mathbf{A} \in \mathbb{C}^{n\times n}$ musíme uvažovat komplexní skalární součin vektorů, pro který platí $\langle \mathbf{e}, \mathbf{a} \rangle = \mathbf{e}^{\mathrm{H}}\mathbf{a}$ [3].

Algoritmus 1 klasický Gram–Schmidtův proces

for $j \leftarrow 1, n$ **do**
  $\mathbf{v_j} = \mathbf{a_j}$
  for $i \leftarrow 1, j-1$ **do**
    $r_{ij} = \mathbf{q}_i^{\mathrm{T}} \mathbf{a}_j$
    $\mathbf{v}_j = \mathbf{v}_j - r_{ij}\mathbf{q}_i$
  $r_{jj} = \mathbf{v}_j^{\mathrm{T}} \mathbf{v}_j$
  $\mathbf{q}_j = \mathbf{v}_j / r_{jj}$

Algoritmus 2 modifikovaný Gram–Schmidtův proces

for $i \leftarrow 1, n$ **do**
  $\mathbf{v_i} = \mathbf{a_i}$
for $i \leftarrow 1, n$ **do**
  $r_{ii} = \mathbf{v}_i^{\mathrm{T}} \mathbf{v}_i$
  $\mathbf{q}_i = \mathbf{v}_i / r_{ii}$
  for $j \leftarrow i+1, n$ **do**
    $r_{ij} = \mathbf{q}_i^{\mathrm{T}} \mathbf{v}_j$
    $\mathbf{v}_j = \mathbf{v}_j - r_{ij}\mathbf{q}_i$

1.2.2 Householderova transformace

Tento postup ortogonalizace je numericky stabilnější než Gram-Schmidtův proces a matice $\mathbf{Q}$ a $\mathbf{R}$ jsou určeny pomocí součinu řady Householderových matic. Householderova matice $\mathbf{H}$ přestavuje v euklidovském zobrazení reflexy vektoru $\mathbf{u}$ a je definována jako

$$\mathbf{H} = \mathbf{I} - 2\frac{\mathbf{u}\mathbf{u}^{\mathrm{T}}}{\mathbf{u}^{\mathrm{T}}\mathbf{u}} = \mathbf{I} - 2\mathbf{v}\mathbf{v}^{\mathrm{T}}. \tag{1.12}$$

Při rozkladu nejprve určíme vektory $\mathbf{u}$ a $\mathbf{v}$ jako

$$\mathbf{u} = \mathbf{x} - \alpha\mathbf{e}_1, \tag{1.13}$$

$$\mathbf{v} = \frac{\mathbf{u}}{\|\mathbf{u}\|}, \tag{1.14}$$

kde $\mathbf{x}$ je sloupcový vektor matice $\mathbf{A}$, $\alpha = \|\mathbf{x}\|$ a $\mathbf{e}_1 = (1, 0, \cdots, 0)^{\mathrm{T}}$ je vektor standardní báze. Provedeme první krok iterace, ve kterém vynulujeme mimo-diagonální prvky prvního sloupce

$$\mathbf{A}_1 = \mathbf{A}, \tag{1.15}$$

$$\mathbf{H}_1 = \mathbf{I} - 2\mathbf{v}\mathbf{v}^{\mathrm{T}}, \tag{1.16}$$

$$\mathbf{H}_1\mathbf{A}_1 = \begin{bmatrix} \alpha_1 & \star & \dots & \star \\ 0 & & & \\ \vdots & & \mathbf{A}_2 & \\ 0 & & & \end{bmatrix}. \tag{1.17}$$

Stejným způsobem opakujeme výpočet pro matici $\mathbf{A}_2$ a tím postupujeme v eliminaci prvků pod hlavní diagonálou, čímž získáme matici $\mathbf{R}$ jako

$$\mathbf{R} = \mathbf{H}_k \cdots \mathbf{H}_2\mathbf{H}_1\mathbf{A}. \tag{1.18}$$

Matici $\mathbf{Q}$ potom určíme jako [1, 4]

$$\mathbf{Q} = \mathbf{H}_1^{\mathrm{T}}\mathbf{H}_2^{\mathrm{T}} \cdots \mathbf{H}_k^{\mathrm{T}}. \tag{1.19}$$

1.2.3 Givensova rotace

V tomto případě používáme pro eliminaci mimo diagonálních prvků matici $\mathbf{G}(i, j, \theta)$, která v euklidovském zobrazení představuje rotaci o úhel θ v rovině i, j.

$$\mathbf{G}(i,j,\theta) = \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & -s & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & s & \cdots & c & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix} \tag{1.20}$$

V případě QR rozkladu matice $\mathbf{A}_{\mathrm{N}\times\mathrm{N}}$ volíme v každém kroku postupně $\mathbf{G}(i, j)$, tak aby platilo $i > 1$, $i > j$ a $i < \mathrm{N}$, a určujeme

$$r = \sqrt{\mathbf{A}(j,j)^2 + \mathbf{A}(i,j)^2}, \tag{1.21}$$

$$c = \mathbf{A}(j,j)/r, \tag{1.22}$$

$$s = -\mathbf{A}(i,j)/r. \tag{1.23}$$

Horní trojúhelníková matice je získána jako

$$\mathbf{R} = \mathbf{G}_{\mathrm{N}-1} \cdots \mathbf{G}_1\mathbf{A}. \tag{1.24}$$

Ortogonální matice je potom dána součinem

$$\mathbf{Q} = \mathbf{G}_1^{\mathrm{T}} \cdots \mathbf{G}_{\mathrm{N}-1}^{\mathrm{T}}. \tag{1.25}$$

Způsobů, kterými určujeme c a s, může být několik, přestože jsou matematicky ekvivalentní, při vyjádření s konečnou přesností mohou některé způsoby vykazovat mnohem lepší numerické výsledky [1]. Ortogonalizace pomocí Givensových rotací je výhodná především v případě matic s menším počtem mimodiagonálních nenulových prvků. Další výhodou je možnost paralelních výpočtu.

1.3 Singulární rozklad

Singulární rozklad, který je v anglické literatuře označovaný jako SVD (Singular Value Decompozition), matice $\mathbf{A} \in \mathbb{C}^{m \times n}$ můžeme zapsat jako

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^{\mathbf{H}}, \tag{1.26}$$

kde $\mathbf{U}_{m \times m}$ a $\mathbf{V}_{n \times n}$ jsou unitární matice a jejich sloupce tvoří levé respektive pravé singulární vektory. Singulární čísla $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r \geq 0$ matice $\mathbf{A}$ nalezneme na diagonále matice $\mathbf{\Sigma}$. Mezi singulárními hodnotami a vlastními čísli existuje vztah

$$\sigma_i^2 = \lambda_i,$$

kde λ_i je i-té vlastní číslo matice $\mathbf{A}^{\mathrm{H}}\mathbf{A}$. Obdobně pravé singulární vektory jsou vlastními vektory matice $\mathbf{A}^{\mathrm{H}}\mathbf{A}$ a levé singulární vektory jsou vlastními vektory matice $\mathbf{A}\mathbf{A}^{\mathrm{H}}$.

1.3.1 Metody výpočtu

Vzhledem k výše uvedeným souvislostem s vlastními čísly a vektory jsou používané stejné metody. Protože $\mathbf{A}^{\mathrm{H}}\mathbf{A}$ je hermitovská, využíváme metod vhodných pro hermitovské matice. Pro řídké matice volíme QR algoritmus, nejčastěji s diagonalizací pomocí Householderovi transformace (viz kap. 1.2.2) a dokončení s využitím Givensových rozkladů (viz kap. 1.2.3). Pro husté matice se používá Jacobiho metoda (viz kap. 2.3) [7].

2 VLASTNÍ ČÍSLA A VLASTNÍ VEKTORY

Vlastní čísla a vlastní vektory jsou důležitým pojmem, nejen v lineární algebře. Využívají se při řešení systémů diferenciálních rovnic, v teorii řízení jsou důležité při určování stability systémů a např. Google při vyhledávání využívá vlastní čísla v algoritmu pro hodnocení stránek. Matice $\mathbf{A}$ představuje lineární zobrazení a pro její vlastní vektory platí, že jejich směr po transformaci zůstane zachován a pouze jejich velikost se změní v poměru vlastního čísla, což můžeme vyjádřit jako

$$\mathbf{A}\mathbf{u}_i = \lambda_i \mathbf{u}_i, \tag{2.1}$$

kde $\mathbf{u}_i$ je i-tý vlastní vektor $\mathbf{A}$ příslušící vlastnímu číslu λ_i. Vlastní čísla můžeme určit řešením tzv. charakteristické rovnice

$$\det(\mathbf{A} - \lambda\mathbf{I}) = 0, \tag{2.2}$$

kde $\mathbf{I}$ je jednotková matice. Vlastní vektor $\mathbf{u}_i$ získáme dosazením příslušného vlastního čísla do rovnice

$$(\mathbf{A} - \lambda_i\mathbf{I})\mathbf{u}_i = 0. \tag{2.3}$$

2.1 Důležité vlastnosti

Determinant matice $\mathbf{A}$ je rovný součinu vlastních čísel

$$\det(\mathbf{A}) = \prod \lambda_i. \tag{2.4}$$

Stopa matice $\mathbf{A}$ je rovna součtu vlastních čísel

$$\text{Stopa}(\mathbf{A}) = \sum \lambda_i. \tag{2.5}$$

Vlastní vektory příslušné různým vlastním číslům jsou navzájem lineárně nezávislé. Hermitovská matice má vlastní čísla reálná. Hermitovská pozitivně (semi)definitní matice má vlastní čísla reálná a kladná (nezáporná).

2.2 Analytické řešení

Z výše uvedených rovnic vyplývá, že pro nalezení vlastních čísel a vektorů $\mathbf{A}_{\mathrm{N}\times\mathrm{N}}$ je nutné nejprve určit $\det(\mathbf{A} - \lambda\mathbf{I})$, čímž získáme polynom $p(\lambda)$ stupně N, jehož kořeny představují hledaná vlastní čísla. Kořeny polynomu do stupně 4 lze získat

algebraicky, v případě vyšších stupňů polynomu využíváme některou z numerických metod. Uvažujme hermitovskou matici $\mathbf{A} \in \mathbb{C}^{4\times4}$ a označme prvky

$$\mathbf{A} = \begin{bmatrix} k & \overline{o} & \overline{r} & \overline{t} \\ o & l & \overline{p} & \overline{s} \\ r & p & m & \overline{q} \\ t & s & q & n \end{bmatrix}. \tag{2.6}$$

Charakteristický polynom můžeme zapsat jako

$$p(\lambda) = \begin{vmatrix} k-\lambda & \overline{o} & \overline{r} & \overline{t} \\ o & l-\lambda & \overline{p} & \overline{s} \\ r & p & m-\lambda & \overline{q} \\ t & s & q & n-\lambda \end{vmatrix} \tag{2.7}$$

a následně vyjádřit jako polynom

$$\begin{aligned} p(\lambda) = \lambda^4 + (-k\ -l-m-n)\lambda^3 + (kl - |p|^2 - |q|^2 - |r|^2 - |s|^2 - |t|^2 - |o|^2 + \\ +km+kn+lm+ln+mn)\lambda^2 + (l|t|^2 + m|s|^2 + n|r|^2 + m|t|^2 + k|p|^2 + \\ k|q|^2 + m|o|^2 + l|q|^2 + n|o|^2 + k|s|^2 + l|r|^2 + n|p|^2 - op\overline{r} - pq\overline{s} \\ -os\overline{t} - qr\overline{t} - r\overline{op} - s\overline{pq} - t\overline{os} - t\overline{qr}) - klm - kln - kmn - lmn)\lambda \\ |o|^2|q|^2 - mn|o|^2 + |p|^2|t|^2 - kn|p|^2 - kl|q|^2 + |r|^2|s|^2 - ln|r|^2 \\ -km|s|^2 - lm|t|^2 + klmn - t\overline{opq} + kpq\overline{s} + nop\overline{r} + lqr\overline{t} \\ +mos\overline{t} - opq\overline{t} + ks\overline{pq} + nr\overline{op} + lt\overline{qr} + mt\overline{os} - os\overline{qr} - qr\overline{os} - pt\overline{rs} - rs\overline{pt}. \end{aligned} \tag{2.8}$$

2.3 Jacobiho metoda

Jacobiho metoda pro výpočet vlastních čísel a vektorů je jednou z nejstarších metod, která je použitelná pro symetrické matice $\mathbf{A}_{\mathrm{N}\times\mathrm{N}}$. Základem je matice rotace $\mathbf{R}_{\mathrm{N}\times\mathrm{N}}$, která je v každém kroku iterace aplikována na matice $\mathbf{D}_{\mathrm{N}\times\mathrm{N}}$ a $\mathbf{V}_{\mathrm{N}\times\mathrm{N}}$. Na konci algoritmu jsou vlastní čísla matice $\mathbf{A}$ umístěny na diagonále matice $\mathbf{D}$ a vlastní vektory jsou tvořeny sloupci v matici $\mathbf{V}$, která je dána součinem rotačních matic. Tento postup lze zapsat jako

$$\mathbf{D}_0 = \mathbf{A}, \tag{2.9}$$

$$\mathbf{D}_{(k+1)} = \mathbf{R}_k^{\mathrm{T}}\mathbf{D}_k\mathbf{R}_k, \tag{2.10}$$

$$\mathbf{V}_{(k+1)} = \mathbf{V}_k\mathbf{R}_k, \tag{2.11}$$

$$\mathbf{V} = \mathbf{R}_1 \cdots \mathbf{R}_k. \tag{2.12}$$

Transformační matice $\mathbf{R}$ je unitární a nemění tedy vlastní čísla a vektory a v každém kroku je volena tak, aby postupně převedla matici $\mathbf{D}$ na diagonální. Označujeme ji pomocí indexů jako

$$\mathbf{R}_{pq} = \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & r_{pp} & \cdots & r_{pq} & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & r_{qp} & \cdots & r_{qq} & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix}. \tag{2.13}$$

Matice rotace představuje při zobrazení v euklidovském prostoru rotaci o úhel θ a prvky matice $\mathbf{R}$ jsou

$$r_{pp} = c = \cos\theta, \qquad r_{pq} = s = -\sin\theta, \tag{2.14}$$

$$r_{qp} = s = \sin\theta, \qquad r_{qq} = c = \cos\theta. \tag{2.15}$$

V praktickém případě určujeme prvky podle toho, který mimo-diagonální prvek bude rotace eliminovat, pomocí vztahů

$$\frac{c^2 - s^2}{sc} = \frac{a_{pp} - a_{qq}}{a_{qp}}, \tag{2.16}$$

$$\beta = \frac{a_{pp} - a_{qq}}{2a_{qp}}, \tag{2.17}$$

$$t^2 + 2\beta t - 1 = 0. \tag{2.18}$$

Z důvodu stability volíme řešení

$$t = \frac{\operatorname{sign}(\beta)}{|\beta| + \sqrt{\beta^2 + 1}}, \tag{2.19}$$

$$\cos\theta = r_{pp} = r_{qq} = \frac{1}{\sqrt{t^2 + 1}}, \tag{2.20}$$

$$\sin\theta = r_{qp} = -r_{pq} = r_{pp}t. \tag{2.21}$$

V připadě hermitovských matic musíme uvažovat úhly θ_1 a θ_2, což vede na

$$\mathbf{R} = \mathbf{R}_{pq}(\theta_2)\mathbf{R}_{pq}(\theta_1), \tag{2.22}$$

$$\theta_1 = \frac{2\phi_1 - \pi}{4}, \tag{2.23}$$

$$\theta_2 = \frac{\phi_2}{2}, \tag{2.24}$$

pro které platí

$$\tan\phi_1 = \frac{\mathrm{Im}\{\mathbf{D}_{p,q}\}}{\mathrm{Re}\{\mathbf{D}_{p,q}\}}, \tag{2.25}$$

$$\tan\phi_2 = \frac{2|\mathbf{D}_{p,q}|}{\mathbf{D}_{p,p} - \mathbf{D}_{q,q}}. \tag{2.26}$$

A výsledná rotační matice je dána jako [8, 10, 1]

$$\mathbf{R} = \mathbf{R}_{pq}(\theta_2)\mathbf{R}_{pq}(\theta_1) = \begin{cases} \delta_{m,n} & m,n \neq p,q, \\ -ie^{-i\theta_1}\,\sin\theta_2 & m = p \text{ a } n = p, \\ -e^{+i\theta_1}\,\cos\theta_2 & m = p \text{ a } n = q, \\ e^{-i\theta_1}\,\cos\theta_2 & m = q \text{ a } n = p, \\ +ie^{+i\theta_1}\,\sin\theta_2 & m = q \text{ a } n = q. \end{cases} \tag{2.27}$$

2.4 QR algoritmus

Jedná se o iterativní algoritmus, kterým opět získáváme matici $\mathbf{D}$ s vlastními čísly matice $\mathbf{A}$ na diagonále a matici $\mathbf{V}$, která je tvořena vlastními vektory matice $\mathbf{A}$. V každém kroku je prováděn QR rozklad, viz 1.2, a postup můžeme zapsat jako

$$\mathbf{D}_0 = \mathbf{A}, \tag{2.28}$$

$$\mathbf{V}_0 = \mathbf{I}, \tag{2.29}$$

$$\mathbf{Q}_k\mathbf{R}_k = \mathbf{D}_k, \tag{2.30}$$

$$\mathbf{D}_{(k+1)} = \mathbf{R}_k\mathbf{Q}_k, \tag{2.31}$$

$$\mathbf{V}_{(k+1)} = \mathbf{V}_k\mathbf{Q}_k. \tag{2.32}$$

2.5 Metoda Rayleigho podílu

Na rozdíl od předchozích metod se jedná o metodu, která pracuje s prvotním odhadem vlastního čísla případně vektoru a postupnými kroky je nalezen právě nejbližší vlastní vektor a příslušné vlastní číslo. Tato metoda předpokládá hermitovskou matici a poměrně rychle konverguje (kubická konvergence). Příbuzné metody pro obecné matice jsou mocninná a inverzní mocninná metoda, které konvergují pomaleji. Určíme počáteční odhad a dále postupujeme podle vztahů

$$\mathbf{b}_{i+1} = \frac{(\mathbf{A} - \mu_i\mathbf{I})^{-1}\mathbf{b}_i}{||(\mathbf{A} - \mu_i\mathbf{I})^{-1}\mathbf{b}_i||}, \tag{2.33}$$

$$\mu_{i+1} = \frac{\mathbf{b}_i^{\mathrm{H}}\mathbf{A}\mathbf{b}_i}{\mathbf{b}_i^{\mathrm{H}}\mathbf{b}_i}. \tag{2.34}$$

Pro implementaci je tedy nutné v každém kroku určit inverzi matice nebo řešit

$$(\mathbf{A} - \mu_i \mathbf{I})\mathbf{c}_{i+1} = \mathbf{b}_i \tag{2.35}$$

a výsledek dále normovat, čímž získáme vektor

$$\mathbf{b}_{i+1} = \frac{\mathbf{c}_{i+1}}{|\mathbf{c}_{i+1}|}. \tag{2.36}$$

3 MOORE-PENROSOVA PSEUDOINVERZE

Jedná se o zobecnění pojmu inverze pro singulární a obdélníkové matice. Pseudoinverzní matice je určena jednoznačně a pokud srovnáme úlohu inverzní a pseudoinverzní matice z hlediska řešení systému rovnic, dostáváme nejlepší řešení vzhledem k podmínce nejmenších čtverců. Mějme matici $\mathbf{A}_{\mathrm{N}\times\mathrm{M}}$ a její pseudoinverzi $\mathbf{X}_{\mathrm{M}\times\mathrm{N}}$, pro kterou platí

$$\mathbf{AXA} = \mathbf{A}, \tag{3.1}$$

$$\mathbf{XAX} = \mathbf{X}, \tag{3.2}$$

$$(\mathbf{AX})^{\mathrm{H}} = \mathbf{AX}, \tag{3.3}$$

$$(\mathbf{XA})^{\mathrm{H}} = \mathbf{XA}. \tag{3.4}$$

Pseudoinverzní matici označujeme jako $\mathbf{A}^{+} = \mathbf{X}$ a jednotlivé způsoby, jakými tuto matici můžeme určit jsou uvedeny dále.

3.1 Metoda singulárního rozkladu

Vlastnosti a způsoby výpočtu tohoto rozkladu jsou popsány v části 1.3 a pseudoinverzi pomocí tohoto rozkladu můžeme určit jako

$$\mathbf{A}^{+} = \mathbf{V}_r\mathbf{\Sigma}_r^{-1}\mathbf{U}_r^{\mathrm{H}}, \tag{3.5}$$

kde

$$\mathbf{A} = \mathbf{U\Sigma V}^{\mathrm{H}} = [\mathbf{U}_r|\mathbf{U}_0]\left[\begin{array}{c|c} \mathbf{\Sigma}_r & 0 \\ \hline 0 & 0 \end{array}\right][\mathbf{V}_r|\mathbf{V}_0]^{\mathrm{H}} = \mathbf{U}_r\mathbf{\Sigma}_r\mathbf{V}_r^{\mathrm{H}}. \tag{3.6}$$

3.2 Výpočet pomocí QR rozkladu

Pro pseudoinvezní matici platí

$$\mathbf{A}^{+} = (\mathbf{A}^{\mathrm{H}}\mathbf{A})^{-1}\mathbf{A}^{\mathrm{H}}, \tag{3.7}$$

$$\tag{3.8}$$

a pomoci QR rozkladu určíme

$$\mathbf{A}^{\mathrm{H}}\mathbf{A} = (\mathbf{QR})^{\mathrm{H}}(\mathbf{QR}) = \mathbf{R}^{\mathrm{H}}\mathbf{Q}^{\mathrm{H}}\mathbf{QR} = \mathbf{R}^{\mathrm{H}}\mathbf{R}. \tag{3.9}$$

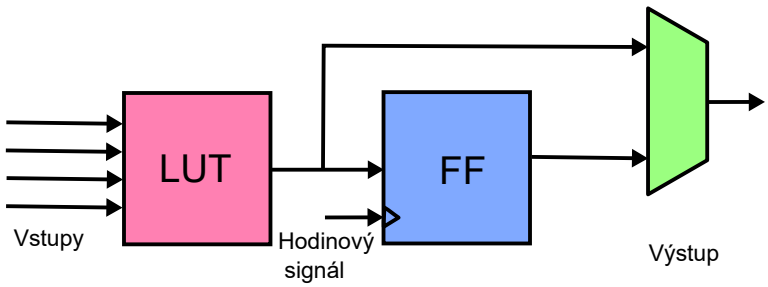
4 METODY PROGRAMOVÁNÍ HRADLOVÝCH POLÍ

V úvodu této kapitoly je uveden obecný popis FPGA obvodů a kroky procesu implementace pomocí tradičních nástrojů. Dále je zde popsáno jakým způsobem do tohoto procesu zapadají nástroje HLS (*High Level Synthesys*).

Následuje detailní popis prostředků, které HLS poskytuje, včetně knihovních funkcí, které jsou používány při výpočtech vlastních čísel a vlastních vektorů a výpočtu pseudoinverze. Pozornost je věnována postupu, kterým lze ovlivnit vlastnosti generovaných RTL designů a to vzhledem k požadavkům na časování, využití prostředků FPGA a rozhraní vstupních a výstupních portů.

4.1 Obecný popis FPGA obvodu

Programovatelná hradlová pole označovaná zkratkou FPGA (*Field-Programmable Gate Array*), jsou integrované obvody programovatelné pomocí HDL jazyků, které se používají při návrhu digitálních integrovaných obvodů ASIC (*Application-Specific Integrated Circuit*). Výroba ASIC je ekonomická při velkých počtech odebraných součástek, oproti FPGA dosahují mnohem lepších vlastností (výkon, spotřeba, plocha). FPGA jsou tvořena ze struktury buněk obsahující generátory logické funkce (kombinační logika) LUT a klopné obvody D neboli FF (*Flip-Flop* - sekvenční logika). Základní FPGA buňka je zobrazena na obr. 4.1. Spojení buněk zajišťuje propojovací struktura, která má za úkol přivedení signálů z jednotlivých buněk a rozvod hodinového signálu. Hodinový signál je generován pomocí smyček fázového závěsu PLL (*Phase locked loop*) a synchronizován po celé ploše obvodu. FPGA obsahují další bloky se specifickými funkcemi jako paměťové BRAM, výpočetní DSP bloky, vstupně výstupní bloky a *transceivery*. Poslední co může FPGA obvod poskytovat jsou integrovaná IP jádra, kterými mohou být celé mikroprocesory, obvody pro kryptografii, paměťová nebo síťová rozhraní. Zjednodušenou strukturu FPGA můžeme vidět na obr. 4.2. Díky vysoké propustnosti mají FPGA široké uplatnění zejména ve zpracování signálu, hrají důležitou roli při vývoji ASIC obvodů a začínají se používat jako výpočetní jednotky superpočítačů nebo datových center.



Obr. 4.1: Základní FPGA buňka

4.2 Opecný postup FPGA designu

4.2.1 Klopný obvod D - FF (*Flip-Flop*)

Jak již bylo naznačeno v úvodu, FPGA obvod je tvořený strukturou buněk, které jsou synchronizovány společným hodinovým signálem. Klopné obvody D neboli *Flip-Flops*, přepínají svůj stav právě při náběžné nebo sestupné hraně hodinového signálu. Slouží jako registry pro vstupy a výstupy kombinační logiky. Důležité je dodržet požadavky na časování a udržet konstantní hodnotu vstupního signálu po určitou dobu před a po příchodu hrany hodinového signálu.

4.2.2 LUT (*Lookup Table*)

Jedná se o blok, který je tvořen rychlými paměťovými buňkami a používá se jako generátor logické funkce, který má vstupní porty (typicky 4 až 6) a jeden nebo více výstupních portů podle konkrétní řady obvodu. Kromě obecné logické funkce, která je buňkám přiřazena, je lze použít pro tvorbu posuvných registrů, nebo rychlých paměťových bloků.

4.2.3 DSP (*Digital Signal Procesing*) blok

Jedná se o speciální výpočetní buňku tvořenou akumulátorem a násobičkou. Její struktura a možnosti konfigurace se liší dle typu obvodu, obecně je navržena nejčastější operace při zpracování signálů (konvoluce, FFT) obdobně jako instrukce MAC (*multiply and acummulate*) v DSP procesorech.

4.2.4 BRAM Bloky

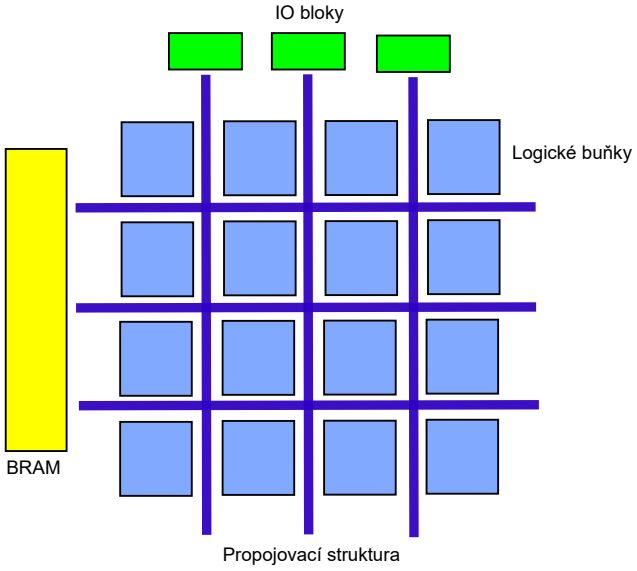
BRAM Bloky jsou konfigurovatelné dvouportové paměťové bloky k uchování zpracovávaných dat. FPGA *Xilinx* řady 7 mají buňky o velikosti 36 Kb, které mohou být konfigurována jako dvě nezávislé 18 Kb paměti nebo jako jedna samostatná. Dále lze zvolit jednu z možností datové šířky, které jsou 64K×1 (v kaskádě se sousedním blokem), 32K× 1, 16K×2, 8K×4, 4K×9, 2K×18, 1K×36, nebo 512×72.

4.2.5 Vstupně výstupní bloky

Vstupně výstupní bloky mají vstupy a výstupy, které jsou vyvedeny na pouzdro FPGA a slouží k připojení vstupních a vstupních signálů. Pro jednotlivé vývody FPGA volíme směr, napěťové standardy, impedanční přizpůsobení, proudové zatížení, diferenční páry, vložené zpoždění a další detaily popsané v příručce [19]. Komunikaci FPGA s okolím mohou zajišťovat také GTX/GTH *transceivery* dosahující přenosových rychlostí od 500 Mb/s do 12,5 Gb/s pro GTX and 13,1 Gb/s GTH, velmi široké možnosti konfigurace těchto *transceiverů* jsou popsány v příručce [20].

4.2.6 Propojovací struktura

Propojovací struktura je složena z propojení jednotlivých sousedních buněk v rámci tzv. *slice* bloků a programovatelného propojení pomocí tzv. *switch matrices* mezi vzdálenějšími bloky. Uživatelská příručka [15] obsahuje detailní popis struktury a jednotlivých buněk pro FPGA obvody *Xilinx Series 7*.



Obr. 4.2: Zjednodušená struktura FPGA

4.3 Obecný postup FPGA designu

Návrh architektury

Jedná se o formální návrh, který popisuje požadavky, rozděluje architekturu na dílčí částí a definuje jejich vstupy, výstupy a chování.

RTL design

Jednotlivé navržené bloky návrhu jsou popsány pomocí HLD jazyků. V současnosti se pro RTL návrh používají jazyky VHDL, Verilog, SystemVerilog, nebo SystemC.

Verifikace RTL návrhu

Ověření pomocí různých technik a nástrojů zda bloky splňují požadované vlastnosti plynoucí z návrhu. Nejčastěji se vytváří referenční model, vstupy ověřovaného a referenčního bloku jsou pak buzeny stejnými stimuly a výstupy vzájemně porovnávány. Existuje však celá řada dalších technik a speciálních metodik pro ověřování funkcionality RTL návrhu.

Syntéza

Převádí RTL popis na tzv. *netlist*, kterým je definována výsledná digitální realizace, čímž se dostáváme na *gate-level*. Při programování FPGA je *netlist* následně zpracován při procesu implementace, stejný *netlist* může být použit jako podklad pro výrobu digitálních ASIC obvodů.

Implementace

Netlist je transformován do zapojení pro cílový obvod, jednotlivým buňkám jsou přiřazeny jejich funkce a je definováno jejich propojení, tento proces se dále člení na fáze *translate*, *map* a *place and route*. V první fázi *translate* je *netlist* přeložen do formátu, který doplňuje odhad zpoždění při přepínání. Ve fázi *map* jsou prvkům z *netlistu* přiřazeny konkrétní prostředky daného FPGA obvodu LUT, FF (*Flip-Flops*), BRAM a další specifické prostředky. Nejnáročnější fází je *place and route*, ve které jsou konkrétní buňky rozmístěny a propojeny.

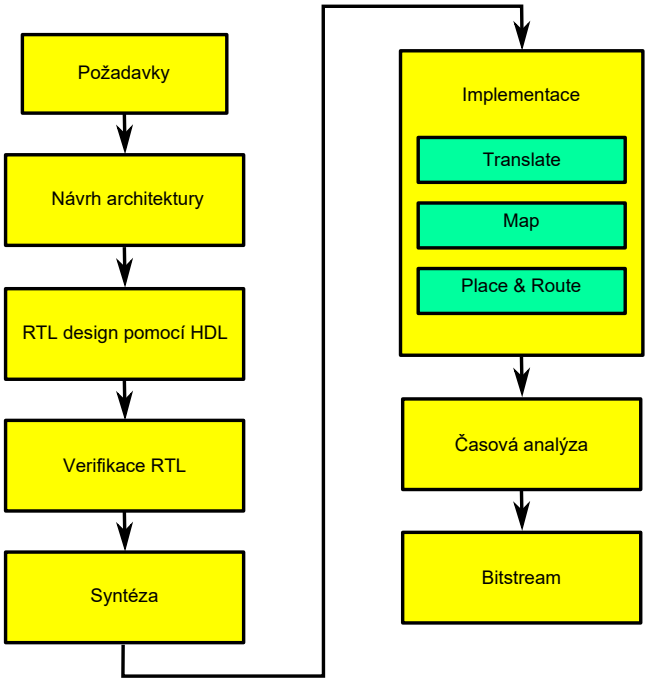
Časová analýza

V této části se ověří, zda zapojení splňuje podmínky časování. Jsou určena zpoždění v propojovací struktuře a kombinační části, součet zpoždění nesmí překročit podmínky dané časováním, pokud navržené zapojení splňuje podmínky máme určitou rezervu časování označovanou jako *timing margin*, případně jako *positive slack*,

pokud vznikne *negative slack*, znamená to, že nejsou splněny podmínky časování a musíme se vrátit k RTL návrhu.

Bitstream

Výsledkem Implementace je *bitstream*, který se přivede do cílového obvodu a ten provádí konfiguraci svých jednotlivých buněk. Celý postup FPGA návrhu je shrnut na obr. 4.3. Detailní informace o procesu syntézy a implementace poskytují uživatelské příručky *Xilinx Vivado* [17, 16].



Obr. 4.3: Proces programování FPGA od návrhu po *bitstream*

4.4 UltraFast Design Methodology

Při postupu, který je uveden výše se vývojáři nejvíce času zabývají implementačními detaily RTL a jeho ověřováním, jedná se o časově náročný proces nejen z hlediska vlastního návrhu a programování ale i z hlediska procesů syntézy a simulací. Opravy a změny na RTL úrovni zabírají mnohem více času než samotný návrh a inovace se těžko zapracovávají, například při přesunu na nové a rychlejší obvody je často nutné přepsat podstatnou část HDL souborů. Jako řešení těchto problémů se snaží *Xilinx* nabídnout nástroje, které souhrnně označuje jako *High-Level Productivity Design*

Methodology, mezi které patří i HLS. Detailně pojednává o tomto konceptu příručka [13].

Výběr a návrh IP bloků

Xilinx poskytuje širokou škálu IP jader, které využívají standardních rozhraní a umožňují vysokou míru konfigurace. V případě nenalezení požadovaných bloků, je možné je vytvořit prostřednictvím HLS. Tvorbě a integraci HLS bloků se podrobně věnuje několik následujících částí.

Integrace a propojení IP bloků

Pro integraci jednotlivých bloků se nejčastěji používá AXI rozhraní, to poskytuje jednotnou formu komunikace a umožňuje volbu několika režimů, na základě charakteru přenosu. Podrobnější popis je v kapitole týkající se rozhraní vstupních a výstupních portů 4.5.6.

4.5 HLS

Neustálý vývoj technologií představuje pro společnosti pracující v oblasti FPGA velký tlak na produktivitu, nové standarty v komunikačních technologiích se objevují stále častěji a využívají čím dál komplikovanější algoritmy. Nové FPGA obvody zároveň poskytují čím dál tím vyšší výkon a jsou zajímavou alternativou pro akceleraci výpočtů v celé škále odvětví. Z těchto důvodů vznikla celá řada méně či více úspěšných nástrojů pro urychlení a zjednodušení procesu FPGA návrhu. Jedním z těchto nástrujů je HLS (*High Level Synthesys*) společnosti *Xilinx*, který umožňuje programování FPGA na vyšší úrovni abstrakce, kdy je možné využít programovací jazyk C, C++, SystemC a OpenCL. HLS má jistá omezení, která vyplývají z vlastností FPGA a jsou to především veškerá systémová volání a dynamická alokace paměti. Dále není možné použít prvky standardní knihovny, právě kvůli využívání zakázaných konstrukcí. HLS poskytuje knihovnu s funkcemi a datovými typy, které je možné při implementaci využít, tato knihovna obsahuje funkce optimalizované pro FPGA.

Vstupem pro HLS jsou zdrojové soubory ve výše uvedených programovacích jazycích a *constraints*, které specifikují limitní hodnoty hodinového signálu a jeho odchylku a vlastnosti cílového FPGA obvodu. Výsledný návrh je ovlivněn pomocí tzv. *direktiv*, které specifikují konkrétní podobu implementace části kódu, ten pak může být optimalizovaný z hlediska zpoždění nebo využití prostředků FPGA. Další důležitou součástí jsou *test bench*, které slouží k ověření jednotlivých funkcí implementace, avšak na výslednou podobu nemají vliv.

Mezi výstupy HLS patří soubory s výsledným RTL, kde je možné volit mezi HDL jazyky VHDL (IEEE 1076-2000) a Verilog (IEEE 1364-2001). Tento výsledek je možné zabalit do tzv. IP bloku. Dále jsou to reporty syntézy a výsledky RTL/C kosimulace.

4.5.1 Proces syntézy

Proces transformace zdrojových souborů na výsledný RTL návrh probíhá v několika fázích:

- **Plánování operací**
  Určuje časování operací podle parametrů cílového FPGA obvodu a optimalizačních direktiv.
- **Přiřazení prostředků**
  Přiřadí fyzické prostředky FPGA obvodu jednotlivým naplánovaným operacím.
- **Tvorba řídící logiky**
  Extrahuje řídící logiku a převede na FSM, které implementuje v RTL.

Vstupně výstupní porty bloků jsou v HLS definovány pomocí argumentů tvz. *top-level* funkce. Dalším funkcím jsou přiřazeny příslušné RTL bloky. Celý proces je řízen pomocí uživatelem specifikovaných *constraints* a *direktiv*. Jednotlivé varianty výsledného návrhu je možné porovnat na základě reportu, který poskytuje tyto informace:

- **Využití prostředků**
  Informace o celkovém počtu použitých FF, LUT, DSP48 a BRAM.
- **Zpoždění**
  Počet hodinových cyklů pro výpočet výstupních parametrů funkce.
- ***Initiation interval***
  Počet hodinových cyklů, po které je blokován vstup funkce pro nová data.
- **Zpoždění smyčky**
  Počet hodinových cyklů, potřebných pro jednu iteraci dané smyčky.
- ***Initiation interval* smyčky**
  Počet hodinových cyklů, po které je blokován vstup smyčky před zpracováním nových dat.
- **Celkové zpoždění smyčky**
  Počet hodinových cyklů, potřebných pro všechny iterace dané smyčky.

Výsledky syntézy jsou v adresáři *syn*, Report je umístěn v podadresáři *report* a výsledné RTL soubory v podadresářích: *verilog*, *vhdl*, *systemc*, podle toho, který RTL výstupní formát je zvolen.

4.5.2 Struktura projektu

V programech v jazyce C je v hierarchii nejvýše hlavní funkce `main()`. Ve *Vivado* HLS, je funkce nejvýše v hierarchii nazývána *top-level* funkce, která je zvolena pro syntézu a může být jedinečná v rámci projektu. *Test bench*, který ve funkci `main()` volá ověřované funkce s různými vstupními argumenty a porovnává návratové hodnoty s referenčními výsledky. Úspěšný *test bench* vrací návratovou hodnotu 0. Příklad *test bench* je v příloze A.7. V jednom projektu je možné vytvořit několik verzí RTL, ty se nazývají *solutions* a mohou se lišit v *constraints* a volbou *direktiv*. Ty mohou být specifikovány přímo ve zdrojových souborech pomocí volání makra preprocesoru `pragma` nebo jako příkazy v souboru `directives.tcl`.

4.5.3 Knihovny *High-level synthesis*

Knihovny HLS obsahují funkce a datové typy, které jsou implementované s důrazem na efektivní převod do RTL. Výsledný RTL návrh je možné optimalizovat pomocí *template* tříd a předáním konfigurace, pomocí vhodně zvolených typů, případně vhodnou volbou direktiv. Knihovna *Xilinx Vivado* HLS obsahuje:

- běžné celočíselné datové typy,
- datové typy pro čísla v pevné řádové čárce s definovatelnou přesností,
- běžné typy v plovoucí řádové čárce v základní a dvojité přesnosti (viz *binary32* a *binary64* IEEE754-2008[18]),
- datový typ pro čísla v plovoucí řádové čárce s 16 bity (*half-precision*),
- funkce matematických operací,
- funkce pro práci s obrazem,
- funkce IP bloků pro FFT a FIR.

Mezi matematickými operacemi můžeme najít celou skupinu funkci pro práci s maticemi.

Formáty s pevnou řádovou čárkou

Celočíselné nativní typy jazyka C mají typicky zarovnání na 8 bitů. V případě RTL návrhu mohou mít sběrnice různou šířku, při použití HLS můžeme specifikovat počet bitů datového typu použitím `u/int` z `ap_cint.h`. Pokud využijeme standardních typů jazyka C, tak proces syntézy povede na méně efektivní RTL návrh. FPGA struktury, které obsahují hardwarové ALU jednotky s pevnou šířkou, v případě bloků DSP48 to může být násobička 25×18 nebo 48-bit akumulátor. Pokud zvolíme nižší šířku použitého datového typu, můžeme snížit využití prostředků FPGA až trojnásobně, jak je ukázáno v příkladu od výrobce [21], kde srovnává filtraci signálu při použití datového formátu s plovoucí desetinnou čárkou a návrh využívající

ap_fixed<18,1> pro koeficienty, ap_fixed<27,15> pro data a ap_fixed<48,19> pro uložení mezivýsledků. Zároveň může být výrazně sníženo zpoždění. Základní HLS datové typy s pevnou řádovou čárkou pro C/C++ jsou v tabulce 4.1, mimo těchto také existují varianty pro jazyk SystemC. Počet bitů datového typu je volen

Tab. 4.1: HLS formáty s pevnou desetinnou čárkou

Jazyk	Datový typ	Hlavičkový soubor
C	u/int<W>	#include "ap_cint.h"
C++	ap_u/int<W>	#include "ap_int.h"
C++	ap_u/fixed<W,I,Q,O,N>	#include "ap_fixed.h"

pomocí parametru W nebo postfixu N (u/intN např.: int22 nebo uint13). Volba neznaménkového typu je možná prefixem u. Maximální hodnota počtu bitů je omezena na 1024 bitů, avšak může být rozšířena až na 32768 pomocí makra AP_INT_MAX_W v ap_int.h . V případě typu s pevnou desetinnou čárkou můžeme ovlivnit jeho vlastnosti pomocí parametrů W, I, Q, O a N, jejich význam a hodnoty jsou uveden v tabulce 4.2.

Tab. 4.2: Parametry ap_u/fixed<>

W	Počet bitů
I	Celá část
Q	Kvantizace
O	Přetečení
N	Bity saturace

Formáty s plovoucí řádovou čárkou

Kromě klasických *float* a *double* datových typů s plovoucí řádovou čárkou definovaných standardem IEEE 754-2008[18] jako *binary32* a *binary64*, podporuje HLS také *half precision* formát, který má 1 znaménkový bit 5 bitů exponentu a 11 (fyzicky 10) bitů mantisy. Tento formát lze importovat z knihovny pomocí #include "hls_half.h", převod mezi formáty např. v případě vstupů nebo výstupů je možný pomocí

value=static_cast<ap_uint<16»(myhalfvalue)

nebo

static_cast<unsigned short>(myhalfvalue).

Další možností je využití pomocné třídy fp_struct<half> a volání metod data() a to_int().

Tab. 4.3: Režimy Zaokrouhlení a přetečení

Režimy Zaokrouhlení	
AP_RND	Zaokrouhlení k plus nekonečnu
AP_RND_ZERO	Zaokrouhlení k nule
AP_RND_MIN_INF	Zaokrouhlení k minus nekonečnu
AP_RND_INF	zaokrouhlení k nekonečnu
AP_RND_CONV	Konvergentní zaokrouhlení
AP_TRN	Odseknutí k minus nekonečnu
AP_TRN_ZERO	Odseknutí k nule
Režimy přetečení	
AP_SAT	Saturace
AP_SAT_ZERO	Saturace k nule
AP_SAT_SYM	symetrická saturace
AP_WRAP	přetečení
AP_WRAP_SM	znamenkove přetečení

Lineární algebra

HLS podporuje práci s maticemi a používá reprezentaci pomocí dvourozměrného pole, v knihovnách HLS je implementována celá řada užitečných funkcí. Od základních operací, jako je násobení matic, po nalezení inverzní matice, přes rozklady matic, kterých může být využito v algoritmech, pro nalezení vlastních vektorů a vlastních čísel matice.

Násobení matic

```
template<
class TransposeFormA,
class TransposeFormB,
int RowsA,
int ColsA,
int RowsB,
int ColsB,
int RowsC,
int ColsC,
typename InputType,
typename OutputType>
void matrix_multiply(
const InputType A[RowsA][ColsA],
const InputType B[RowsB][ColsB],
OutputType C[RowsC][ColsC])
```

Podporované typy jsou :

- ap_fixed
- float
- x_complex<ap_fixed>
- x_complex<float>

QR rozklad

```
template<
bool TransposeQ,
int RowsA,
int ColsA,
typename InputType,
typename OutputType>
void qrf(
const InputType A[RowsA][ColsA],
OutputType Q[RowsA][RowsA],
OutputType R[RowsA][ColsA])}
```

podporované typy pro `InputType` a `OutputType` jsou:

- ap_fixed
- float
- x_complex<ap_fixed>
- x_complex<float>

Singulární rozklad

```
template<
int RowsA,
int ColsA,
typename InputType,
typename OutputType>
void svd(
const InputType A[RowsA][ColsA],
OutputType S[RowsA][ColsA],
OutputType U[RowsA][RowsA],
OutputType V[ColsA][ColsA])
```

Podporované typy jsou :

- float
- x_complex<float>

4.5.4 Simulace

Simulace se ve *Vivado* HLS provádí pomocí *test bench* funkcí, které volají implementované funkce a ověřují jejich výstupní hodnoty. Stejný *test bench* je možné využít pro základní otestování C funkcí před syntézou a následné ověření RTL návrhu po syntéze. Úspěch simulace je reprezentován návratovou hodnotou 0 a detailnější informace je možné vypsat pomocí funkce `printf()`. Výsledky a log simulace jsou umístěny v adresáři *csim*. Tyto procesy jsou označovány jako *C simulation* a *C/RTL co-simulation*.

C *test bench*

Výhodou testování funkcí v jazyce C je mnohem rychlejší ověření implementace než simulace a *test bench* na RTL úrovni. Základním konceptem je předání testovacích hodnot a porovnání výstupů funkcí s předem známým výsledkem. V příloze A.7 je ukázána typická struktura pro *test bench*. Nejprve jsou definovány potřebné proměnné pro aktuální data, načtení testovacích dat, uložení dat z výstupů ověřovaných funkcí a očekávaných hodnot (řádek 4–12). Následně jsou načtena testovací data ze souborů `inA.dat` a `inB.dat`(řádek 12–25). Ve smyčce jsou předána testovací data implementovaných funkcím a uloženy hodnoty výstupu (řádek 26–34). Poté jsou načtena kontrolní data pro ověření a porovnány s výstupy funkcí, v případě neshody je návratová hodnota nastavena na 1. Nakonec je zobrazen výsledek pomocí výpisu do konzole a vracením návratové hodnoty je *test bench* ukončen.

4.5.5 Optimalizace

Optimálních parametrů výsledného RTL návrhu se dá dosáhnout jednak dodržováním zásad ve zdrojovém kódů, které vedou na efektivní implementaci a reflektují vlastnosti FPGA platformy. Druhým způsobem je vhodná volba HLS direktiv, které mají přímý vliv na výsledný návrh.

Jednou z důležitých zásad je omezit počet čtení vstupních parametrů, případě použít buffer s vhodnou velikostí. Pokud je délka pole vysoká, snažíme se opět o minimalizaci přístupů a pokoušíme se s daty nemanipulovat při zpracování a využít menších lokálních záchytných registrů. Stejné pravidlo platí i pro výstupní hodnoty zapisované do výstupních portů, zároveň je dobré vhodně volit počet potřebných výstupních portů. Pokud zpracováváme tok dat, je nejvhodnější použít knihovnu `hls::streams`, která poskytuje potřebné rozhraní a zajišťuje efektivní implementaci. V případě procesů využívajících *pipelining* je vhodnější provádět podmíněné větvení uvnitř, než podmíněné vykonávání větších samostatných úloh.

Optimalizace propustnosti

Pipelining je technika paralelního zpracování, kdy je složitější úloha rozdělena na vzájemně nezávislé kroky, které je možné vykonávat paralelně. Nejednoduší je demonstrovat *pipelining* na procesu rozděleného do 3 fází, které zabírají přibližně stejnou dobu. Např.: čtení dat, zpracování dat a zápis výsledku. *Pipelining* může být využit při zpracování konkrétní funkce případně aplikován při provádění operací uvnitř smyčky. K tomu je určena direktiva `PIPELINE`. Smyčky jsou v této části automaticky rozvinuty, a pokud je *pipelening* součástí další funkce, je jeho využití v její režii. Pokud ho však využívá, bude to mít pozitivní vliv na *pipelening* celku. Výjimkou jsou smyčky s variabilním počtem průchodů, které nemohou být automaticky rozvinuty a pokud necheme přijít o možnost *pipeliingu* musíme se jim vyhnout. Ve smyčce vzniká hluché místo mezi dokončením jednoho cyklu a začátkem nového, tomu může být zabráněno použitím volby `REWIND`. V projektu můžeme využít možnosti automatického *pipelininguu* smyček, ten se automaticky aplikuje pro cykly do zadaného rozsahu průchodů. Vedle počtu hodinových cyklů zpoždění (*latency*) operace je důležitým pojmem *Initial Interval* označovaný zkratkou II, který je roven počtu taktů mezi dokončením aktuální a začátkem nové iterace a prakticky určuje dobu výpočtu. Typickou příčinou pro nedosažení II=1 jsou pole implementovaná pomocí *block RAM*, které mají vždy pouze 2 porty pro čtení a zápis. Řešením je rozdělení pole na více menších pomocí direktivy `ARRAY_PARTITION`, rozdělení má 3 varianty:

- blokové,
- cyklické,
- kompletní.

Další možností konfigurace je provádění automatického rozdělení polí. V základním nastavení je smyčka převedena do RTL návrhu tak, že každá iterace sdílí stejné prostředky. Pomocí direktivy `UNROLL` můžeme smyčku rozvinout a určitý počet iterací bude provedený paralelně. Důležitým aspektem při dobrém návrhu *pipelininguu* je ošetření případně odstranění závislostí pomocí direktivy `DEPENDENCE`. Pokročilejší forma optimalizace se aplikuje direktivou `DATAFLOW` v případě, kdy provádíme sekvenci úkonů a data předáváme z výstupu na vstup následující části. Potom je vytvořen kanál mezi entitami, který dovoluje zahájení operací před dokončení předcházející funkce. Tento postup je omezen právě přímím propojením jednotlivých částí, nelze využít podmíněných odboček, zpětné vazby ani podmíněné ukončení smyček.

Optimalizace pomocí konfigurace

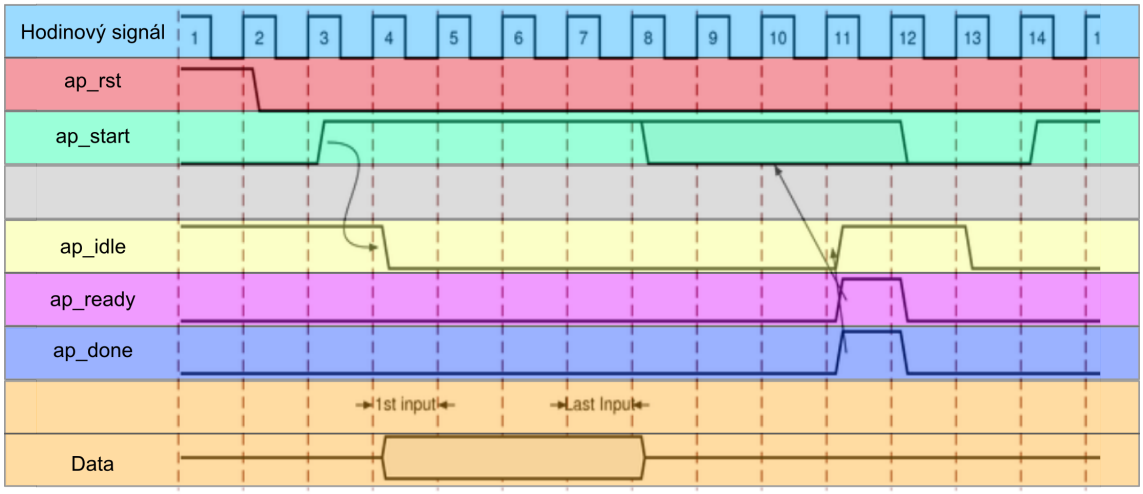
Některé algoritmy zejména z knihovny lineární algebry jsou implementovány pomocí šablonových funkcí, k dispozici je vždy základní varianta algoritmu, kde sou parametrem šablony rozměry a typ matice. Konfigurovatelní varianta s postfixem top přidává další konfigurační parametr. Konfigurace může obsahovat parametry algoritmu (počet provedených iterací), a dále obsahuje parametry, které ovlivňují použitou architekturu, parametry pipeliningu, rozvinutí smyček, tedy parametry, které jsou popsány výše.

4.5.6 Rozhraní vstupních a výstupních portů

Vivado HLS bloky disponují třemi druhy portů:

- *Clock* a *Reset* porty: `ap_clk` a `ap_rst`.
- *Block-Level interface protocol* porty slouží k signalizaci mezi jednotlivými bloky, k dispozici máme `ap_start`, `ap_done`, `ap_ready` a `ap_idle`.
- *Port Level interface protocols* porty jsou definované na základě argumentů *top-level* funkce a přenášejí samotná data.

Úrovně signálů na portech sloužících k signalizaci jsou zobrazeny na obr. 4.4. Chování portů *Block-Level interface protocol* portů je možné ovlivnit direktivou `ITERFACE` a je na výběr z několika formátů rozhraní.



Obr. 4.4: Průběh signalizace HLS bloků

AXI rozhraní

AXI rozhraní je standardní rozhraní pro propojení jednotlivých IP bloků, HLS poskytuje následující varianty:

AXI4-Lite

Toto rozhraní slouží především ke komunikaci IP Jádra s procesorem, umožňuje sloučit několik portů do jednoho rozhraní, včetně portů pro řízení bloku `ap_start`, `ap_done`, `ap_ready` a `ap_idle`. HLS k těmto rozhraním vytváří ovladače, které slouží k řízení bloku z mikroprocesoru. Součástí knihovny ovladačů jsou funkce pro inicializaci komunikace s bloky a funkce pro zápis a čtení jednotlivých portů. Toto rozhraní je dále využíváno při ověření bloků na vývojovém kitu v části 5.4.3, kde jsou popsány vygenerované funkce ovladače a je ukázáno jejich použití.

AXI4 master

Jedná se o rozhraní, které je poskytováváno pro argument typu pole nebo reference k přenosu většího množství dat. Existují dva módy přenosu *Individual* a *Burst*. V prvním případě je ke každému přenesenému prvku vygenerována adresa a jsou zapsána příslušná data. V případě *Burst* je použita bázová adresa a vektor dat, který je přenášen. V HLS je tento přenos umožněn pomocí volání funkce `memcpy()`.

AXI4-Stream

Implementuje rozhraní pro daný argument jako vstupní nebo výstupní *stream*, rozhraní je výhodné pro algoritmy, které zpracovávají určitý tok signálu. Je možné definovat *stream* s postaním kanálem nebo bez. K řízení toku dat jsou k dispozici signály `TVALID` a `TREADY`.

Paměťová rozhraní

Další možnou volbou jsou paměťová rozhraní, následný přenos dat je pak realizován jako zápis nebo čtení z paměti, k dispozici jsou tedy adresní, datové a řídící porty. Na výběr jsou tři možnosti rozhraní:

- `ap_memory`
- `ap_bram`
- `ap_fifo`

4.5.7 Export a generování IP bloku

Výsledné funkce HLS můžeme následně používat, jako součást dalších funkcí pro tvorbu složitějších bloků, nebo můžeme bloky exportovat a použít v prostředí *Vivado* nebo *System Generator for DSP*. Rozhraní, které bloky poskytují byly popsány v předchozí kapitole 4.5.6. *Vivado* HLS poskytuje následující volby:

- **IP Catalog** - exportuje jako IP blok, který je možné využít ve *Vivado* včetně ovladačů pro řízení pomocí procesoru.
- **System Generator for DSP** - exportní formát pro nástroje, které využívají k FPGA návrhu prostředí *Simulink*, které je součástí *Matlab*.
- **Synthesized Checkpoint** - Tento formát provede syntézu vygenerovaných RTL souborů a umožňuje použití bloků v prostředí *Vivado*.

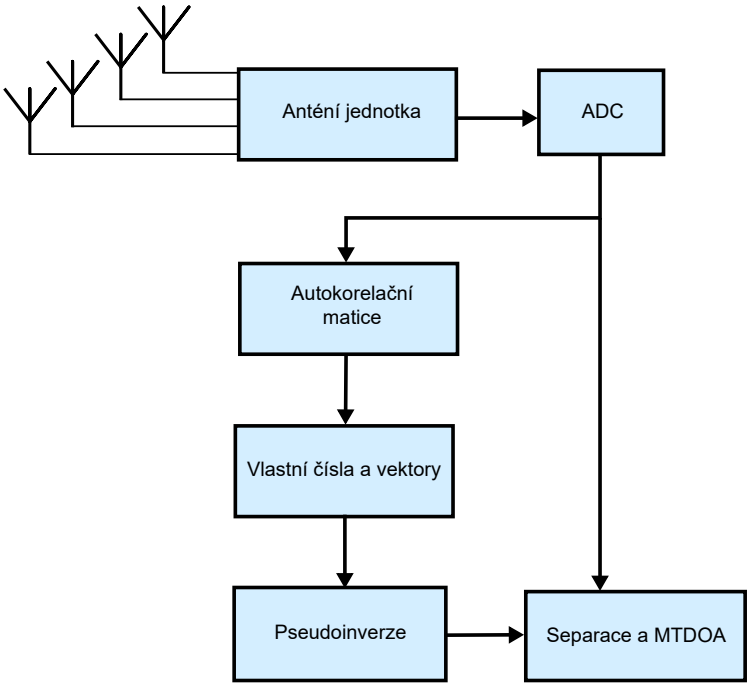
5 IMPLEMENTACE ALGORITMŮ NA HRADLOVÉM POLI

V úvodní částí této kapitoly je popsán obecně systém, ve kterém mají bloky pro výpočet pseudoinverze a výpočet vlastních čísel a vektorů figurovat. Následuje popis struktury jednotlivých variant, která je rozdělena dle použitých algoritmů a parametrů pro optimalizaci implementace. Dále je popsán algoritmus, jakým jednotlivé varianty pracují a kterou dekompozici z úvodních kapitol využívají. U každé varianty jsou uvedeny parametry výsledného FPGA návrhu, numerické vlastnosti a celkové zhodnocení těchto parametrů.

Na závěr jsou popsány možnosti jednotlivých exportních formátů pro další práci s výstupními bloky a varianty rozhraní vstupních a výstupních portů pro komunikaci s okolím. Na praktickém příkladu, jsou ověřeny jednotlivé bloky a je popsán způsob zapojení a přenos dat.

5.1 Obecný popis systému

Následuje obecný popis systému, ve kterém navrhované bloky figurují, systém lze klasifikovat, jako tzv. MIMO (*Multiple-input multiple-output*) systém, ve kterém figuruje více vysílacích antén transpondérů letadel a více přijímacích antén multikanálového přijímače. Vstupem do systému zobrazeného na blokovém schématu na obr. 5.1 jsou přijaté vzorky signálu, segment signálu tvoří matici, kde jednotlivé řádky představují komplexní signál z jednoho anténního prvku. Ze signálu získáme odhad autokorelační matice segmentu, z této matice pak určujeme vlastní čísla a vlastní vektory. Vlastní vektory podle určitého pravidla sestavujeme do matice pro výpočet pseudoinverze. Pseudoinverze je pak aplikována na vzorky signálu, čím získáváme nové vektory signálů, které jsou následně zpracovány. Alternativní způsob implementace vypouští odhad autokorelační matice, provádí rozklad přímo ze signálu a následně aktualizuje hodnoty matice až k výsledným vlastním číslům a vektorům. Výsledky slouží k odhadu směru příchodu na základě fázových poměrů na jednotlivých anténách a k separaci směsi vzniklé smíšením více zpráv. Implementace těchto výpočtů je náročná, protože vyžaduje operace s maticemi v komplexní aritmetice. Z těchto důvodů byl zvolen nástroj HLS, který tyto operace podporuje.



Obr. 5.1: Blokové schéma systému

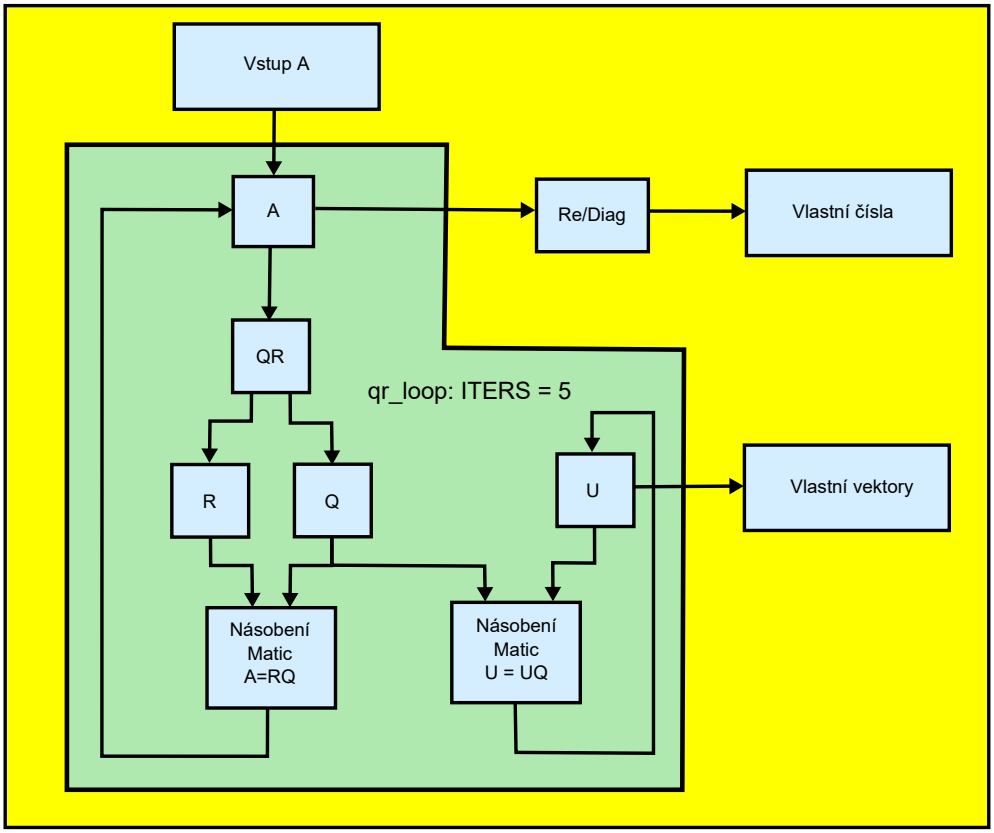
5.2 Implementace výpočtu vlastních čísel a vektorů

Výpočet vlastních čísel a vektorů byl implementován pomocí nástrojů HLS. Základem implementace jsou funkce z knihovny lineární algebry, které provádí dekompozice matic a další pomocné maticové operace. Důležitým prvkem je volba konfigurace pro použité funkce. Výsledkem jsou jednotlivé varianty IP Bloků pro výpočet vlastních čísel a vektorů. Vlastnosti jednotlivých variant a jejich srovnání je uvedeno v následujících částech této kapitoly.

5.2.1 Varianta s QR dekompozicí – `eig_qr`

Varianty `eig_qr` využívají pro výpočet vlastních čísel a vektorů QR algoritmus, který je popsaný v částí 2.4, který v každé iteraci provádí QR rozklad viz část 1.2. Tento rozklad je prováděn pomocí knihovní funkce HLS, která využívá pro rozklad Givensovy rotace viz 1.2.3. Parametry, se kterými funkce pracuje jsou definovaný v hlavičkovém souboru, mezi ně patří rozměry matice, definice datového typu matice a konfigurace QR rozkladu. Jako datový typ prvků matice uvažujeme komplexní číslo s plovoucí desetinnou čárkou v základní přesnosti, čili *float* v aktuálním standardu

IEEE 754-2008 [18] označovaný jako *binary32*. Dále je zde definován počet iterací, což je zásadní parametr, který ovlivňuje výslednou numerickou přesnost a dobu výpočtu. Prezentované výsledky uvažují 5 iterací. Na základě tohoto algoritmu vznikly varianty RTL, které se odlišují použitou optimalizací, která je dosažena pomocí direktiv a zmíněné konfigurace funkce `hls:qr_top` viz tab. 5.1 s popisem parametrů a tab. 5.2 s parametry jednotlivých variant.



Obr. 5.2: Blokový diagram QR algoritmu

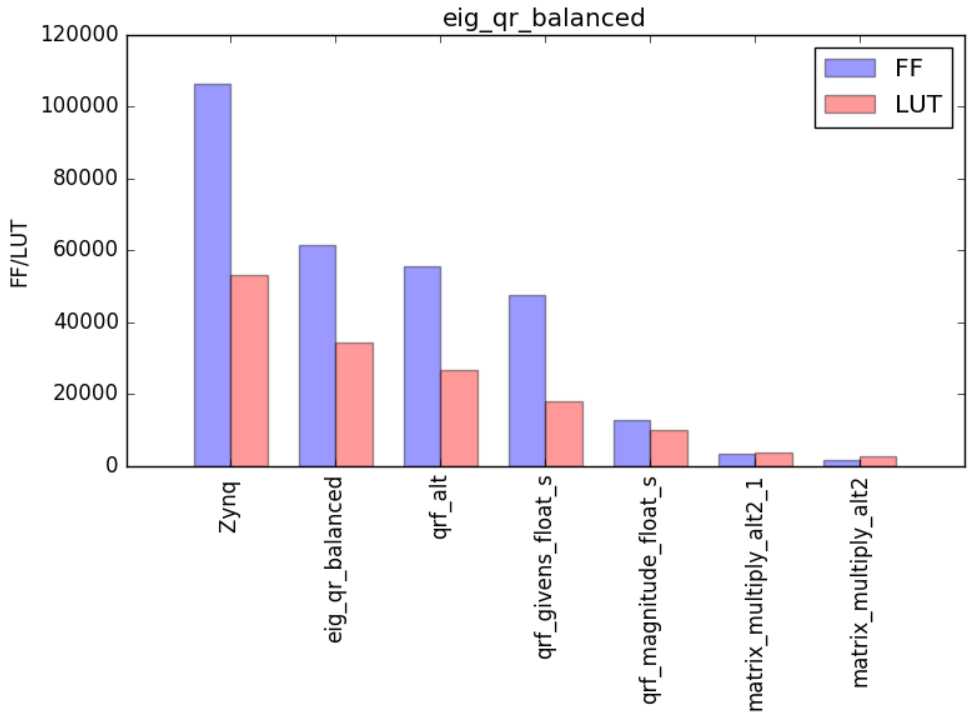
Pro zvolenou variantu `eig_qr_balanced` je v grafu na obr. 5.3 znázorněno detailní využití prostředků FPGA pro rozměr matice 4×4, první sloupec tvoří prostředky kterými disponuje obvod *Zynq 7020*, ve druhém sloupci jsou zobrazeny celkové požadavky této varianty a další sloupce tvoří jednotlivé moduly. Dominantní je v této variantě modul QR dekompozice `qrf_alt`, tento modul využívá modul `qrf_givens` pro výpočet Givensovy rotace, zbývající prostředky využívají moduly pro maticové násobení.

Parametr konfigurace	Def.	Popis parametru
`static const int ARCH`	1	1 – paralelní implementace 0 – základní architektura
`static const int CALC_ROT_II`	1	*Initial Interval* pro výpočet rotace
`static const int UPDATE_II`	4	*Pipelining* pro smyčku rozkladu
`static const int UNROLL_FACTOR`	1	Rozbalení smyčky rozkladu

Tab. 5.1: Konfigurace `hls:qrf_top` funkce pomocí `struct qrf_traits`

Parametr	`balanced`	`small`	`fast`	`faster`
`ARCH`	1	1	1	1
`CALC_ROT_II`	1	8	1	1
`UPDATE_II`	4	8	2	2
`UNROLL_FACTOR`	1	1	1	2

Tab. 5.2: Konfigurace `hls:qrf_top` funkce pomocí `struct qrf_traits`



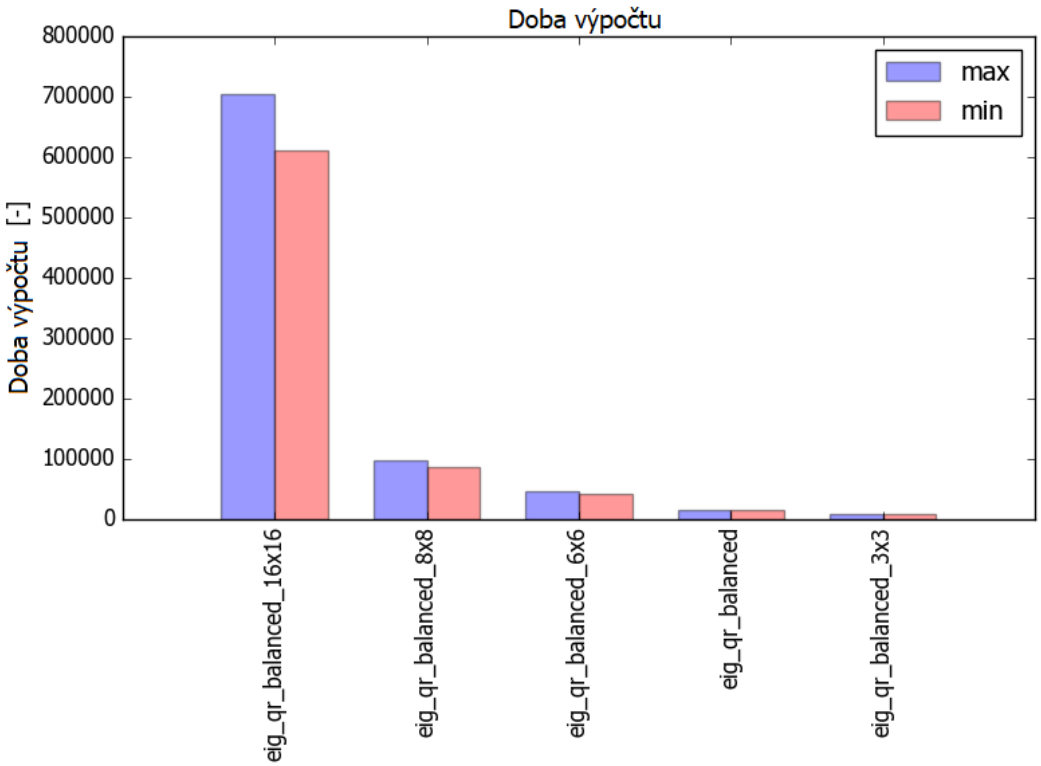
Obr. 5.3: Využití prostředků `eig_qr_balanced` pro rozměr matice 4×4

Závislost výsledných parametrů na rozměru matice

Pro variantu `eig_qr_balanced` byly vytvořeny *soulutions*, které se vzájemně liší pouze rozměrem matice. Z výsledků uvedených v tabulce 5.3 je patrné, že využití prostředků FPGA je na rozměru matice téměř nezávislé a varianty se liší pouze dobou výpočtu. Tato závislost je znázorněna graficky viz obr. 5.4.

Varianta	FF	LUT	DSP	BRAM
`eig_qr_balanced_16x16`	61901	34486	90	24
`eig_qr_balanced_8x8`	61720	34313	90	24
`eig_qr_balanced_6x6`	62208	34473	90	24
`eig_qr_balanced` (4×4)	61688	34327	90	16
`eig_qr_balanced_3x3`	60954	32776	80	16

Tab. 5.3: Závislost využití prostředků na rozměru matice



Obr. 5.4: Závislost doby výpočtu na rozměru matice pro variantu `eig_qr_balanced`

5.2.2 Varianta se singulárním rozkladem – `eig_svd`

varianty `eig_svd` využívají pro výpočet vlastních čísel a vektorů vztah mezi vlastními čísly matice a singulárními hodnotami matice pro hermitovské pozitivně de-

finitní matice. Singulární rozklad je prováděn pomocí knihovní funkce HLS, která využívá Jacobiho metodu. Parametry výsledného RTL jsou definovány v hlavičkovém souboru `eig_svd.h`, opět je zde definován rozměr matic a datový typ, kterým je komplexní číslo s plovoucí desetinnou čárkou typu *binary32* viz IEEE 754-2008 [18]. Výstupní RTL můžeme ovlivnit konfigurací `hls:svd_top`, konfigurace obsahuje parametry pro *pipelining* a parametry pro algoritmus, jako je počet iterací. Detailní popis parametrů je tabulce 5.4 a konkrétní nastavení parametrů pro jednotlivé varianty je v tab. 5.5.

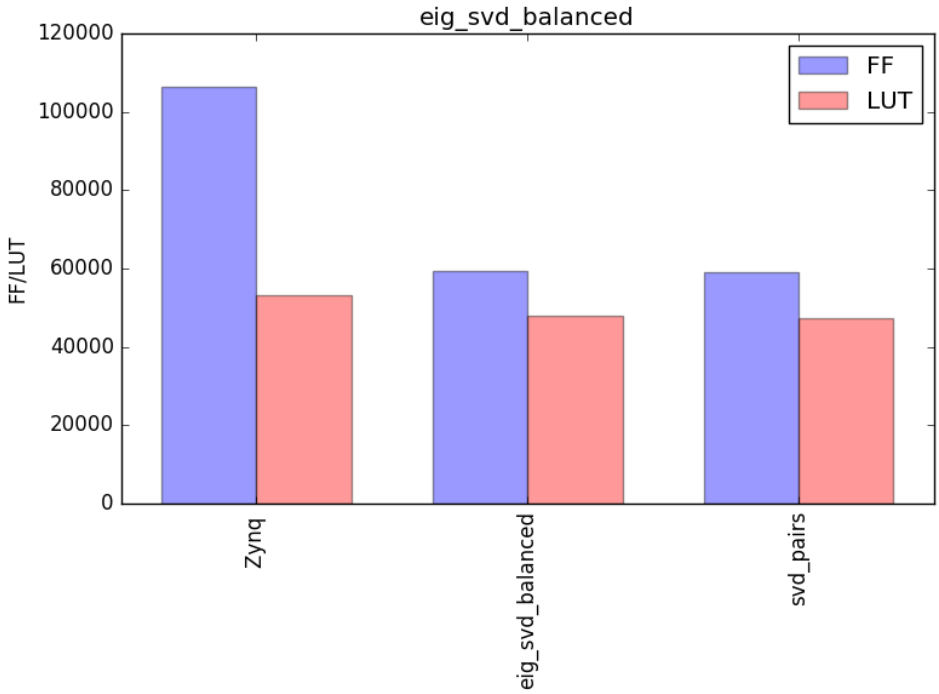
Parametr konfigurace	**Def.**	**Popis parametru**
`static const int NUM_SWEEPS`	10	typicky 6–10
`static const int ARCH`	1	1 – paralelní implementace 0 – základní architektura
`static const int OFF_DIAG_II`	8	*pipelining* pro smyčku mimo diagonálních prvků
`static const int DIAG_II`	8	*pipelining* pro smyčku diagonálních prvků

Tab. 5.4: Konfigurace `hls:svd_top` funkce pomocí `struct svd_traits`

Parametr	`balanced`	`small`	`fast`	`fast_low_iter`
`NUM_SWEEPS`	10	6	10	6
`ARCH`	1	1	1	1
`OFF_DIAG_II`	8	8	4	4
`DIAG_II`	8	8	4	4

Tab. 5.5: Konfigurace `hls:svd_top` funkce pomocí `struct svd_traits`

Opět bylo vygenerováno několik variant lišící se parametry optimalizace, dosažené výsledky časování a využití FPGA nalezneme v části 5.2.3. Pokud porovnáme s předchozí variantou v části 5.5, detailní využití prostředků FPGA pro rozměr matice 4×4 zobrazuje pouze prostředky *top* modulu `eig_svd_balanced` a pod modulu `svd_pairs` pro výpočet singulárního rozkladu, tento výsledek je očekávaný, varianta nevyužívá žádně další funkce z knihovny HLS a na rozdíl od QR rozkladu není modul `svd_pairs` členěný na další pod moduly. Tento modul využívá pro výpočet Jacobiho metodu, což je patrné ze zdrojových souborů knihovny HLS.

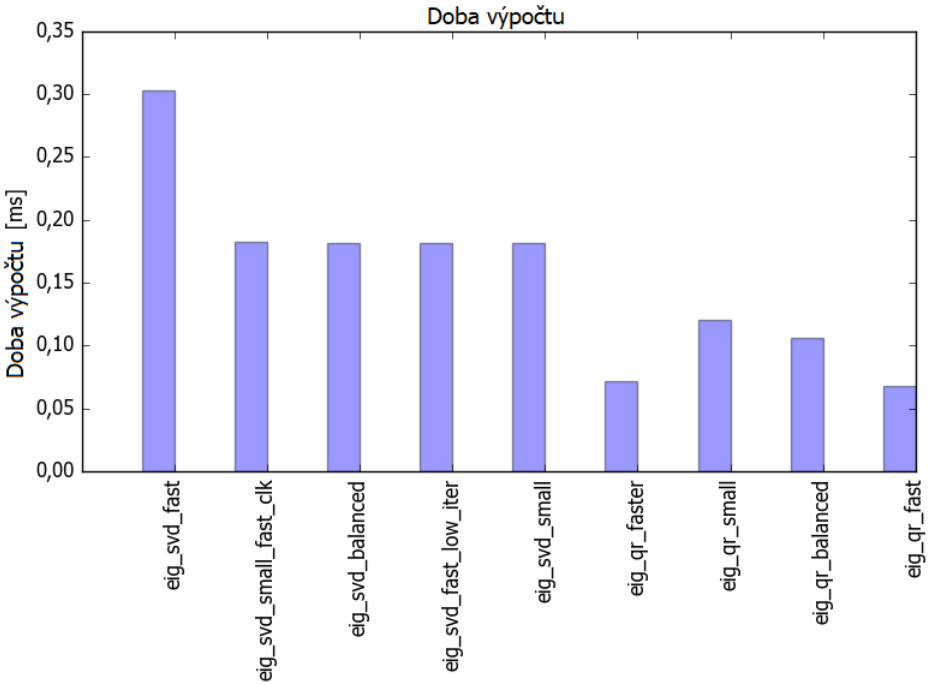


Obr. 5.5: Využití prostředků `eig_svd_balanced` pro rozměr 4×4

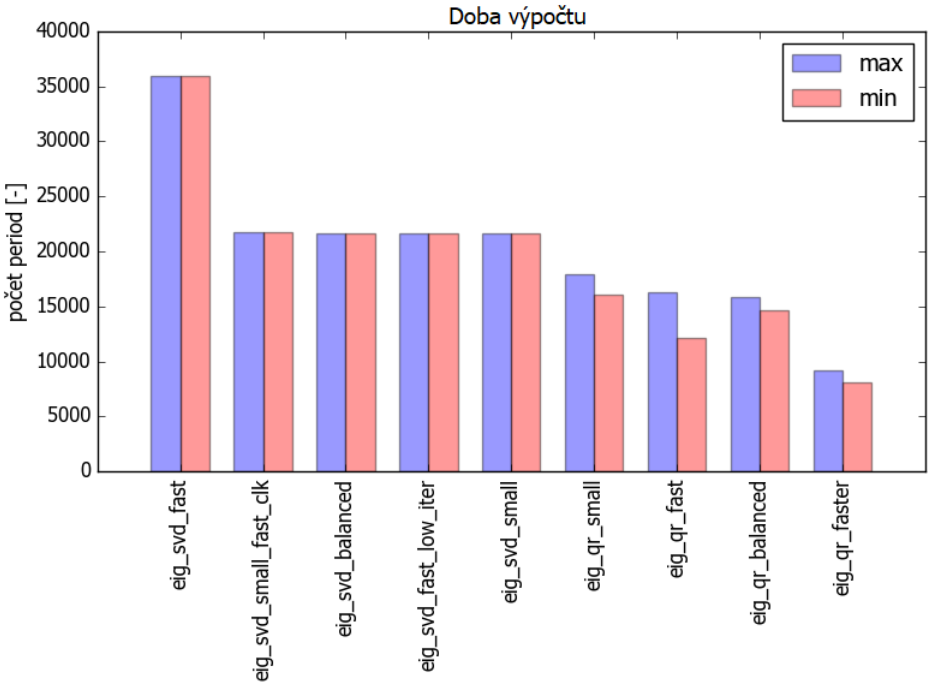
5.2.3 Srovnání variant

Srovnání z hlediska časování

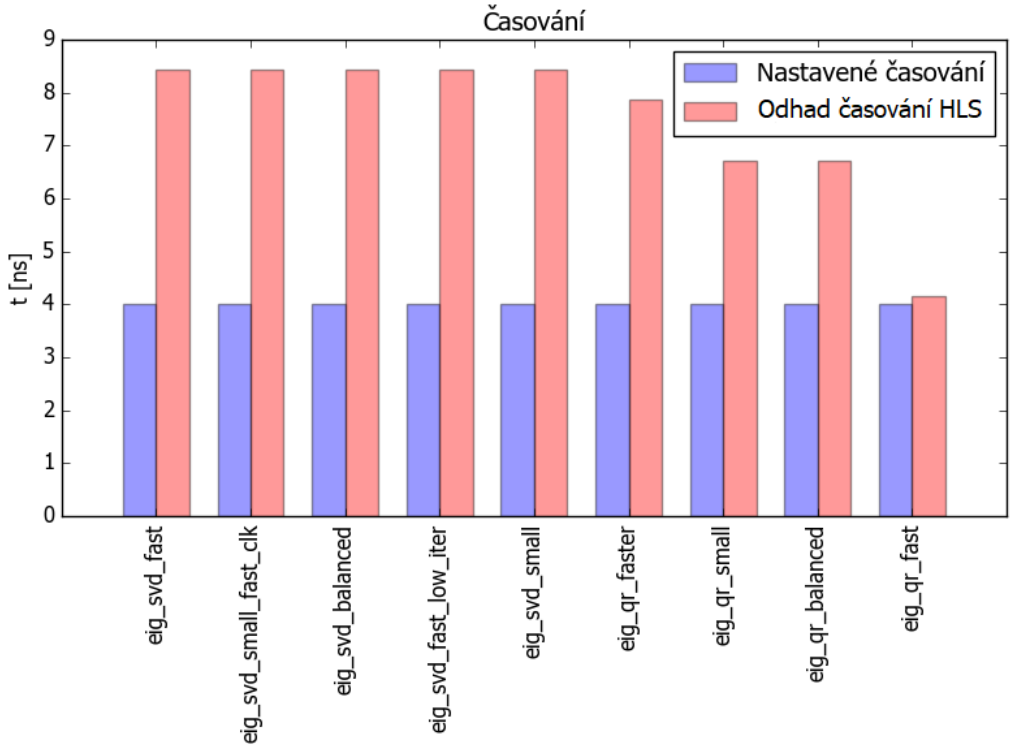
V tabulce 5.6 jsou shrnuty časové parametry jednotlivých variant. Tyto informace jsou získány z reportů vygenerovaných při syntéze ve *Vivado* HLS. Srovnání z hlediska časování je důležité pro zvolení algoritmu pro konkrétní aplikaci. Výsledné parametry jsou závislé na volbě FPGA obvodu při procesu syntézy. Při srovnání jsou uvažovány varianty pracující s komplexními maticemi o rozměrech 4×4 a obvod *Zynq 7020*. Graf na obr. 5.6 porovnává výsledné doby výpočtu v ms získané z výsledků doby výpočtu, na obr. 5.7 jsou zobrazeny minimální a maximální doba výpočtu v počtech hodinových impulzů. Tyto výsledky budou na jednotlivých FPGA obvodech srovnatelné. Poslední graf na obr. 5.8 vykresluje nastavenou a výslednou periodu hodinových pulsů, která je ovlivněna složitostí RTL a vybraným FPGA obvodem. Největší doby výpočtu, tedy nejhorších výsledků dosahuje varianta `eig_svd_fast`, i přes zvolenou optimalizaci dosahuje dobu výpočtu 0,3 ms, na to má vliv především náročný algoritmus singulárního rozkladu. Nejlepšího výsledku dosáhly varianty využívající QR rozklad, konkrétně `eig_qr_fast` a `eig_qr_faster`, dosahující doby výpočtu 0,07 ms, avšak tyto výsledky jsou závislé na počtu iterací QR algoritmu, který je volen z hlediska požadované numerické přesnosti, v tomto případě konkrétně 5 iterací.



Obr. 5.6: Srovnání doby výpočtu vlastních čísel a vlastních vektorů komplexní matice 4×4



Obr. 5.7: Srovnání doby výpočtu vlastních čísel a vlastních vektorů komplexní matice 4×4



Obr. 5.8: Výsledky nastavení časování pro jednotlivé varianty

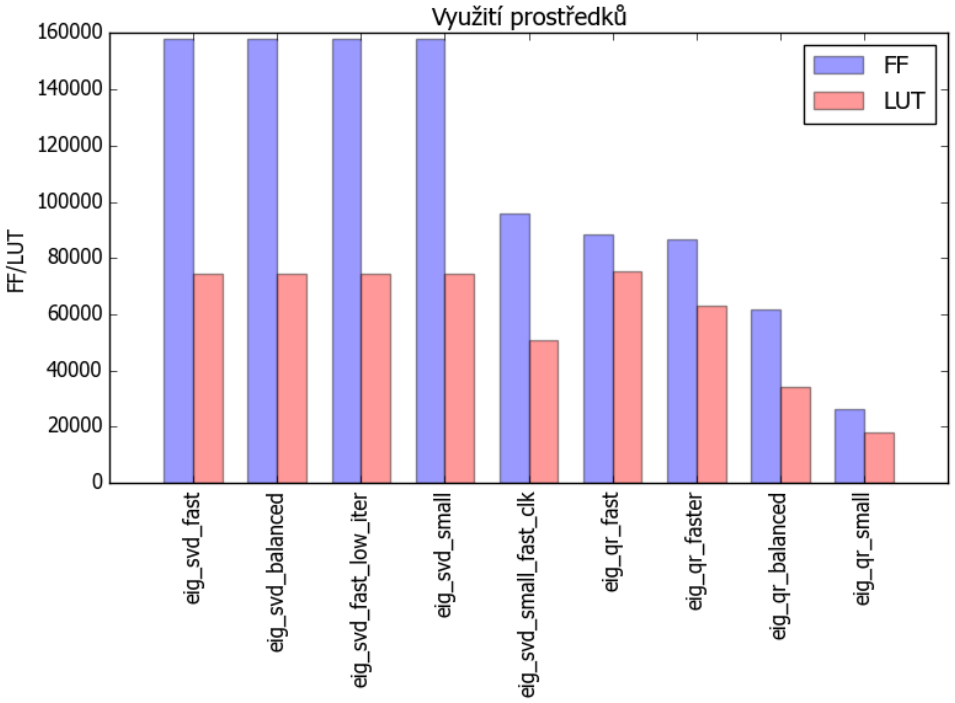
Varianta	**Doba výpočtu** [-]	**Časování** [ns]	**Doba výpočtu** [ms]
`eig_svd_balanced`	36158	8,42	0,304
`eig_svd_fast`	35948	8,42	0,303
`eig_svd_small`	21722	8,42	0,183
`eig_svd_small_fast_clk`	21722	8,42	0,183
`eig_svd_fast_low_iter`	21596	8,42	0,182
`eig_qr_faster`	9155	7,87	0,072
`eig_qr_small`	17951	6,71	0,120
`eig_qr_balanced`	15883	6,71	0,107
`eig_qr_fast`	16279	4,15	0,068

Tab. 5.6: Srovnání jednotlivých variant pro výpočet vlastních čísel a vektorů komplexní matice 4×4

Srovnání z hlediska využití prostředků

V tabulce 5.7 je shrnuto využití prostředků FPGA. Tyto informace jsou získané z reportů vygenerovaných při syntéze ve *Vivado* HLS. Srovnání z hlediska využití

prostředků je důležité pro zvolení vhodné kombinace algoritmu a FPGA obvodu. Při srovnání jsou uvažovány varianty pracující s komplexními matice o rozměrech 4×4. Z grafu na obr. 5.9 je patrné, že nejvyšší nároky na prostředky FPGA má algoritmus využívající singulárního rozkladu ve variantě `eig_svd_fast`. Nejméně prostředků využívá varianta `eig_qr_small`. Ostatní varianty využívají prostředky FPGA srovnatelně a podle zvolené optimalizace se pohybují potřebné počty LUT a FF mezi 50 a 80 tisíci.



Obr. 5.9: Využití prostředků pro výpočet komplexní matice 4×4

Varianta	FF	LUT	DSP	BRAM
`eig_svd_fast`	157962	74670	273	32
`eig_svd_fast_low_iter`	157959	74668	273	32
`eig_svd_balanced`	95929	50984	160	32
`eig_svd_small`	95926	50982	160	32
`eig_svd_small_fast_clk`	95926	50982	160	32
`eig_qr_fast`	88325	75247	170	8
`eig_qr_faster`	86846	63234	230	8
`eig_qr_balanced`	61688	34327	90	16
`eig_qr_small`	26532	18125	44	14

Tab. 5.7: Využití FPGA jednotlivých variant pro výpočet vlastních čísel a vektorů komplexní matice 4×4

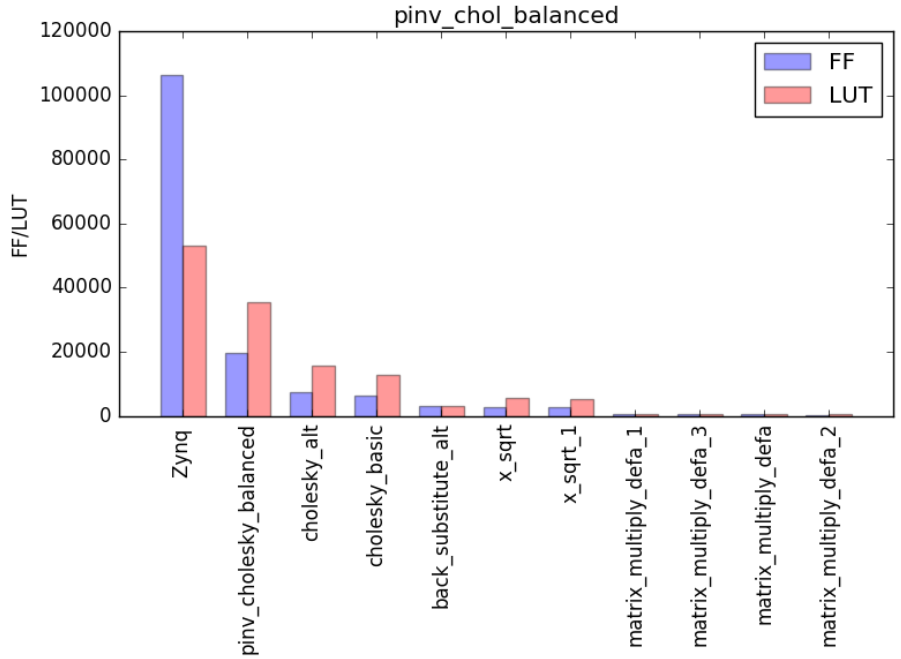
5.3 Implementace výpočtu pseudoinverze matice

5.3.1 Varianta s Choleskeho rozkladem – `pinv_chol`

V případě této `pinv_chol` vycházíme z algoritmu, který využívá vlastností trojúhelníkových matic, které vznikají při Choleského dekompozici viz 1.1.1, odvození těchto vztahů se věnuje článek [9]. Implementace tohoto algoritmu v *Matlab* je součástí příloh. HLS umožňuje implementaci funkce pro Choleskeho dekompozici ve variantě s plovoucí řádovou čárkou,ale i pevnou řádovou čárkou, kterou předchozí varianty neumožňují. Počet bitů pevné a zlomkové částí použitého datového typu lze zvolit v hlavičkovém souboru `pinv_chol.h`, způsob definice takového datového typu je popsán v části 4.5.3. Pro testování byl zvolen typ `hls::x_complex<ap_fixed<16,8»`, který využívá 8 bitů pro celou i zlomkovou část. Parametry výsledného RTL můžeme ovlivnit konfigurací, kterou opět nalezneme hlavičkovém souboru. Pro funkci `hls::cholesky`, obsahuje konfigurace parametry pro *pipelining* uvedené v tabulce 5.8. Grafické znázornění na obr. 5.10 opět poskytuje detailní informace ohledně využití prostředků FPGA této varianty pro rozměr matice 4×4, můžeme si všimnout modulů `cholesky_alt` a `cholesky_basic`, první z nich je využitý při výpočtu dekompozice a druhý slouží k výpočtu inverze. Mezi další bloky patří blok pro výpočet odmocniny a pochopitelně bloky pro násobení matic.

Parametr konfigurace	Def.	Popis parametru
`typedef InputType PROD_T`	–	Typ pro násobení
`typedef InputType ACCUM_T`	–	Typ akumulátoru
`typedef InputType ADD_T`	–	Typ pro sčítání
`typedef InputType DIAG_T`	–	Typ pro diagonální prvky
`typedef InputType RECIP_DIAG_T`	–	Typ pro mimo diagonální prvky
`typedef InputType OFF_DIAG_T`	–	Typ pro mimo diagonální prvky
`typedef OutputType L_OUTPUT_T`	–	Typ výstupní matice **L**
`static const int ARCH`	1	0 – Základní architektura 1 – nízké zpoždění 2 – nejnižší zpoždění
`static const int INNER_II`	1	*pipelining* pro smyčku diagonálních prvků
`static const int UNROLL_FACTOR`	1	Rozbalení smyčky
`static const int UNROLL_DIM`	–	Dimenze rozbalení smyčky
`static const int ARCH2_ZERO_LOOP`	`true`	Nastavení smyčky `CHOLESKY_ALT2`

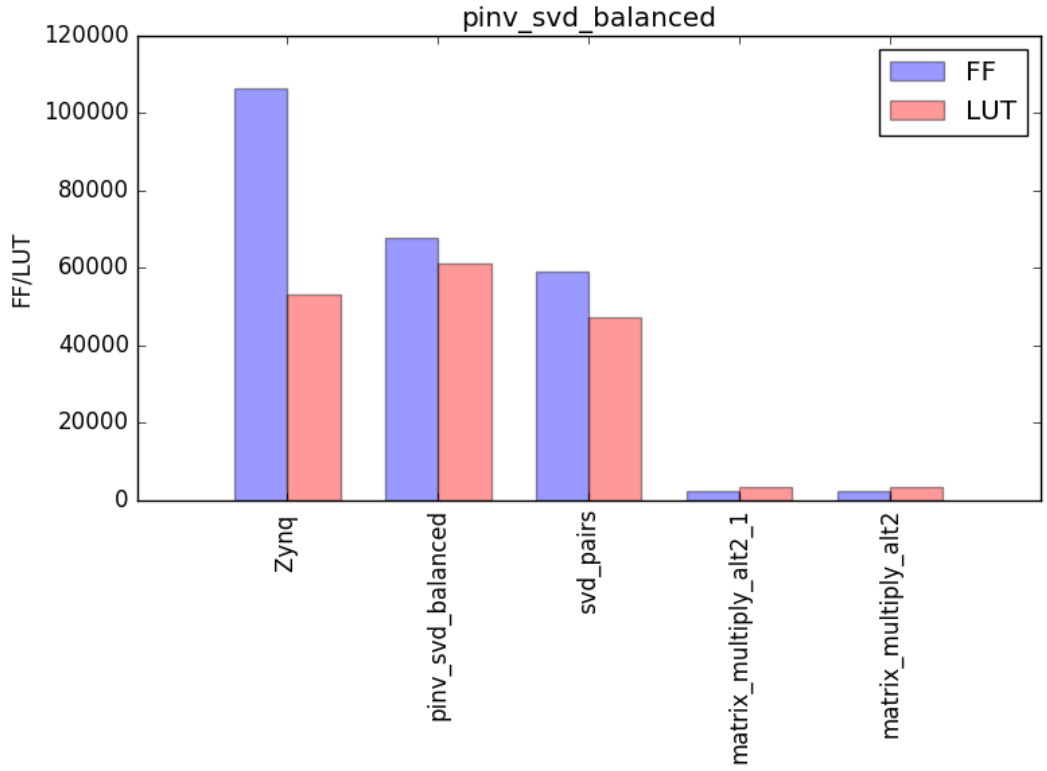
Tab. 5.8: Konfigurace `hls:cholesky_top` funkce pomocí `cholesky_traits`



Obr. 5.10: Využití prostředků varianty `pinv_chol_balanced` pro komplexní matici 4×4 typu `hls::x_complex<ap_fixed<16,8>`

5.3.2 Varianta se singulárním rozkladem – `pinv_svd`

Varianta `pinv_svd` využívá pro výpočet pseudoinverze rozklad na singulární hodnoty viz 1.3. Singulární rozklad je prováděn pomocí knihovní funkce HLS, která využívá Jacobiho metodu. Parametry výsledného RTL můžeme ovlivnit konfigurací `hls:svd_top`, konfigurace byla popsána v části 5.4. Varianta uvažuje komplexní matice typu *binary32* viz IEEE 754-2008 [18]. Rozměry matice, datový typ a konfigurace pro jednotlivé varianty jsou definované v hlavičkovém souboru. Detailní využití prostředků FPGA zobrazuje pouze prostředky *top* modulu `eig_svd_balanced` a pod modulu `svd_pairs` pro výpočet singulárního rozkladu a násobení matic, tento výsledek je očekávaný, varianta nevyužívá žádně další funkce z knihovny HLS a na rozdíl od QR rozkladu není modul `svd_pairs` členěný na další pod moduly. Tento modul využívá pro výpočet Jacobiho metodu, což je patrné ze zdrojových souborů knihovny HLS.



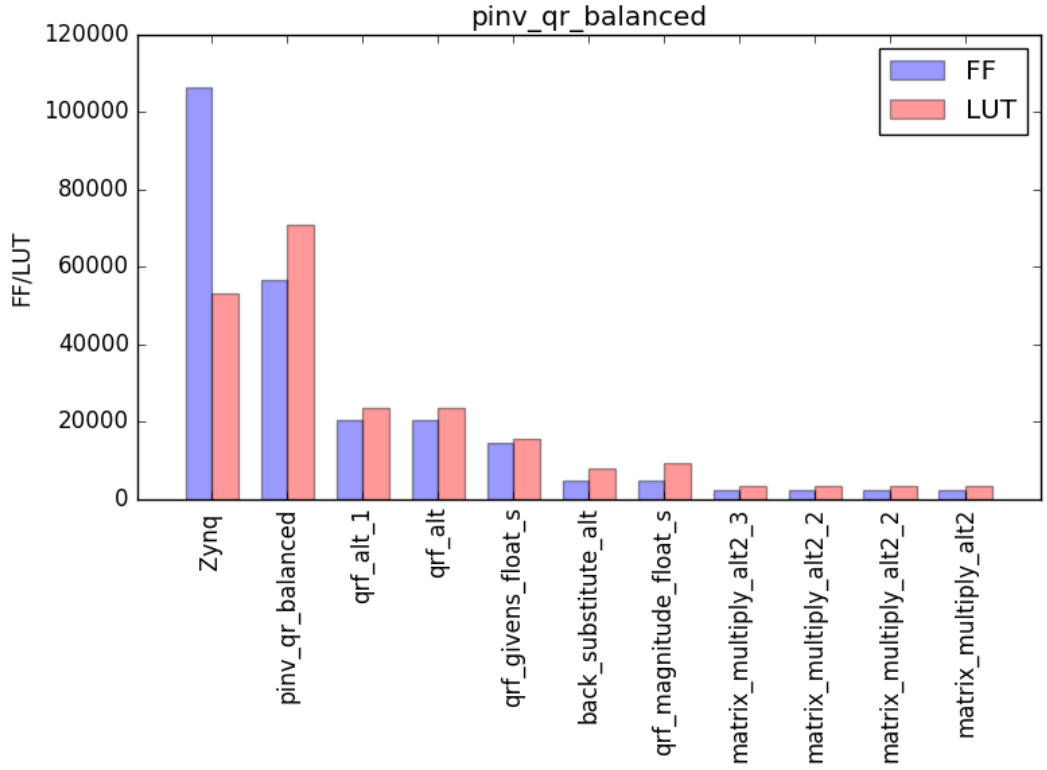
Obr. 5.11: Využití prostředků `pinv_svd_balanced`

5.3.3 Varianta s QR dekompozicí – `pinv_qr`

Varianta `pinv_qr` implementuje algoritmus popsaný v části 3 vycházející ze vztahů mezi maticemi získanými pomocí QR rozklad viz kapitola 1.2. Tento rozklad je

prováděn pomocí knihovní funkce HLS využívající Givensovy rotace viz část 1.2.3. Parametry, se kterými funkce pracuje jsou definovaný v hlavičkovém souboru. Jedná se o rozměry matice, konfiguraci funkce `hls:qr_top` a definici datového typu matice. Konkrétně se jedná o komplexní číslo s plovoucí řádovou čárkou v základní přesnosti viz *binary32* IEEE 754-2008 [18]. Na základě tohoto algoritmu vznikly varianty RTL lišící se použitou optimalizací pomocí konfigurace funkce `hls:qr_top` viz tab. 5.1. Srovnání jednotlivých variant je uvedeno v části 5.3.4.

Pro zvolenou variantu `pinv_qr_balanced` je na na obr. 5.12 graficky znázorněno detailní využití prostředků FPGA pro rozměry matice 4×4. Dominantní je v této variantě modul QR dekompozice `qrf_alt`, tento modul se ve výsledném RTL vyskytuje dvakrát, jednou pro samotnou dekompozici a podruhé pro výpočet klasické inverze. Dále využívá modul `qrf_givens` pro výpočet Givensovy rotace a moduly pro maticové násobení.

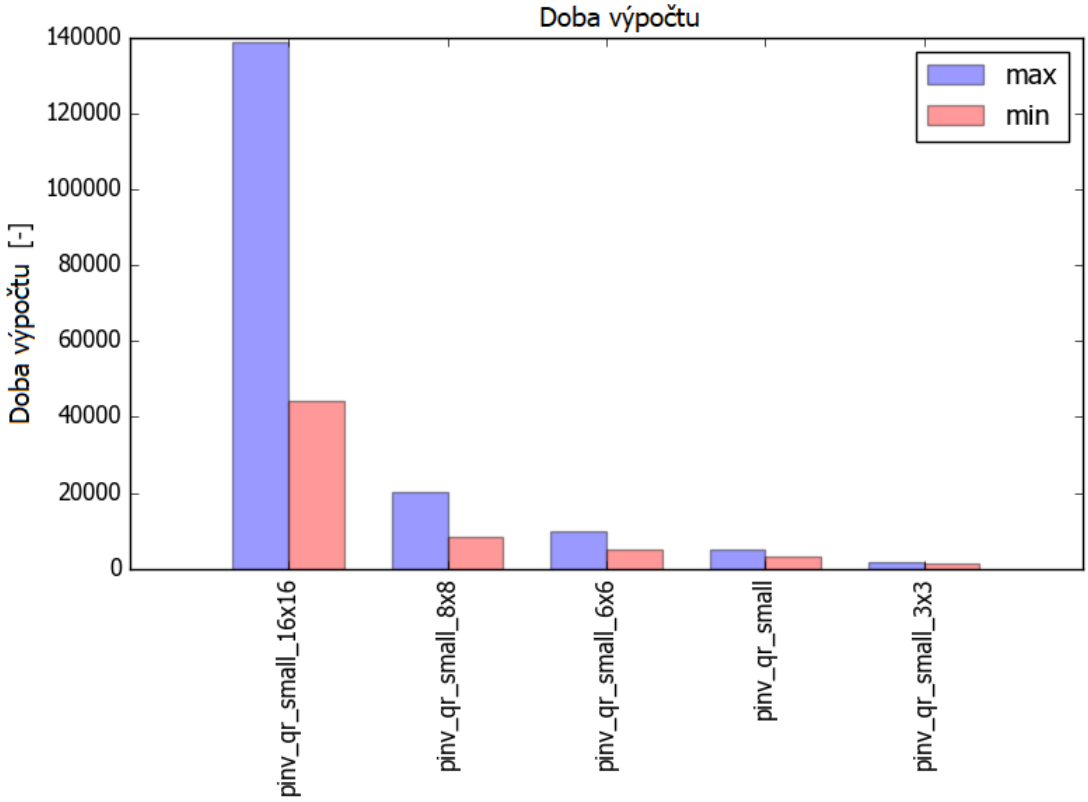


Obr. 5.12: Využití prostředků `pinv_qr_balanced` pro komplexní matice 4×4

Závislost výsledných parametrů na rozměru matice

Pro variantu `pinv_qr_small` byly opět vytvořeny *soulutions*, které se vzájemně liší pouze rozměrem matice. Z výsledků uvedených v tabulce 5.9 je patrné, že využití

prostředků FPGA je na rozměru matice téměř nezávislé a varianty se liší pouze dobou výpočtu. Tato závislost je znázorněna graficky na obr. 5.13.



Obr. 5.13: Závislost doby výpočtu na rozměru matice

Varianta	FF	LUT	DSP	BRAM
`pinv_qr_small_3x3`	39625	41368	116	30
`pinv_qr_small` (4×4)	62024	43724	116	34
`pinv_qr_small_6x6`	38618	41065	116	57
`pinv_qr_small_8x8`	38432	40931	116	57
`pinv_qr_small_16x16`	38711	41287	116	57

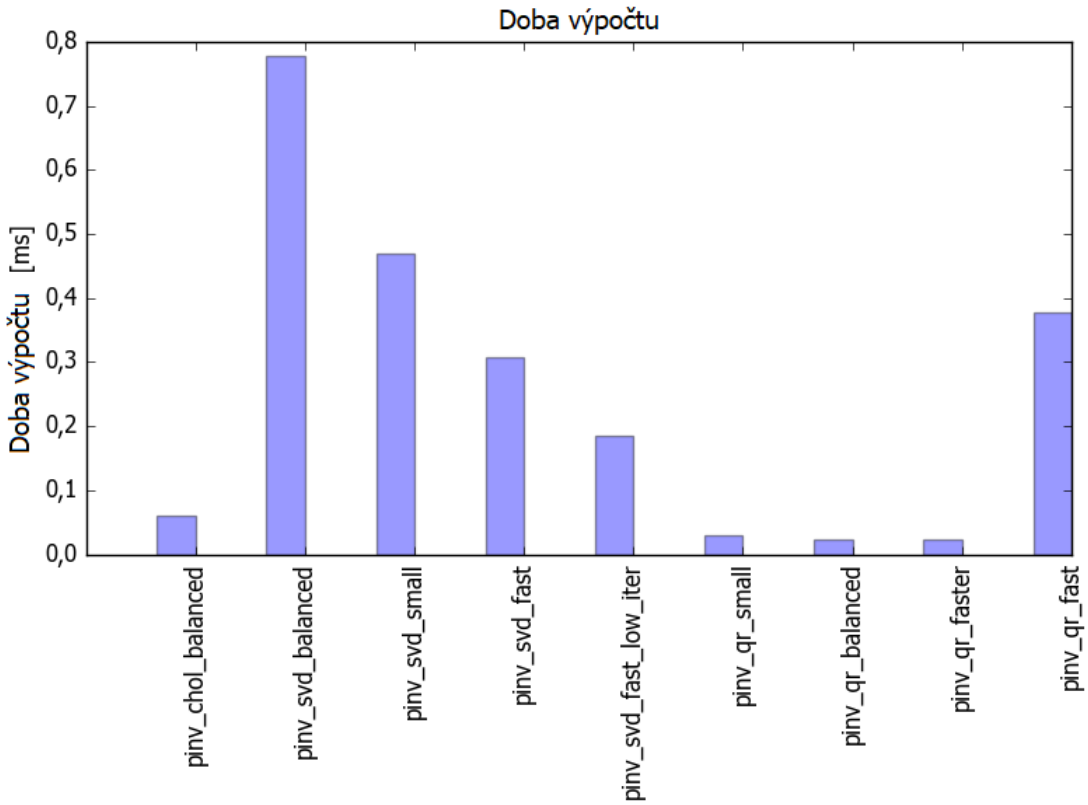
Tab. 5.9: Doba výpočtu v závislosti na rozměru matice

5.3.4 Srovnání variant

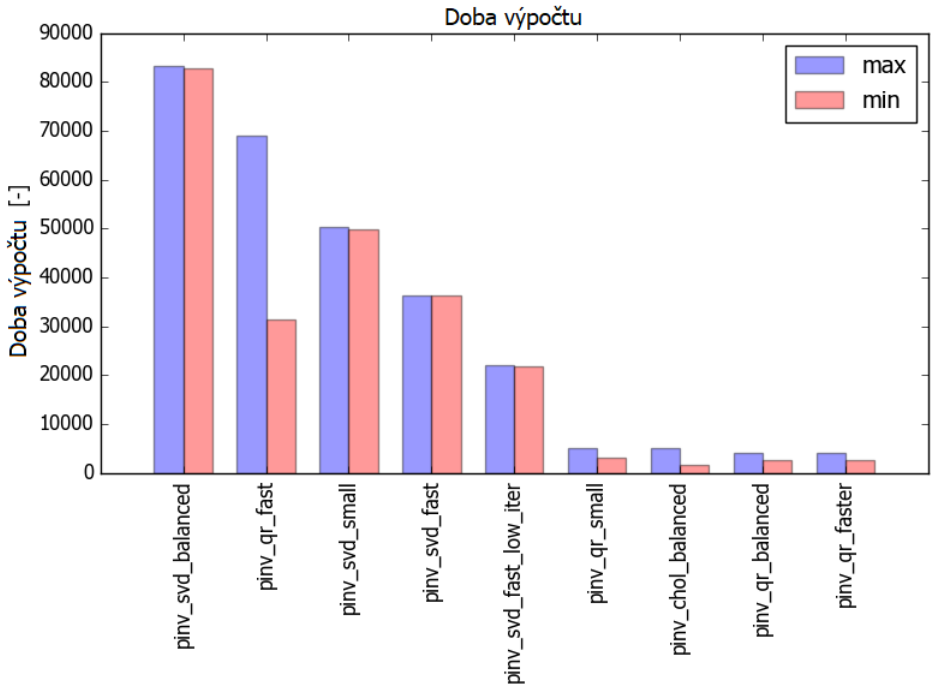
Srovnání z hlediska časování

V tabulce 5.10 jsou shrnuty časové parametry jednotlivých variant. Tyto informace jsou získané z reportů vygenerovaných při syntéze ve *Vivado* HLS. Toto srovnání

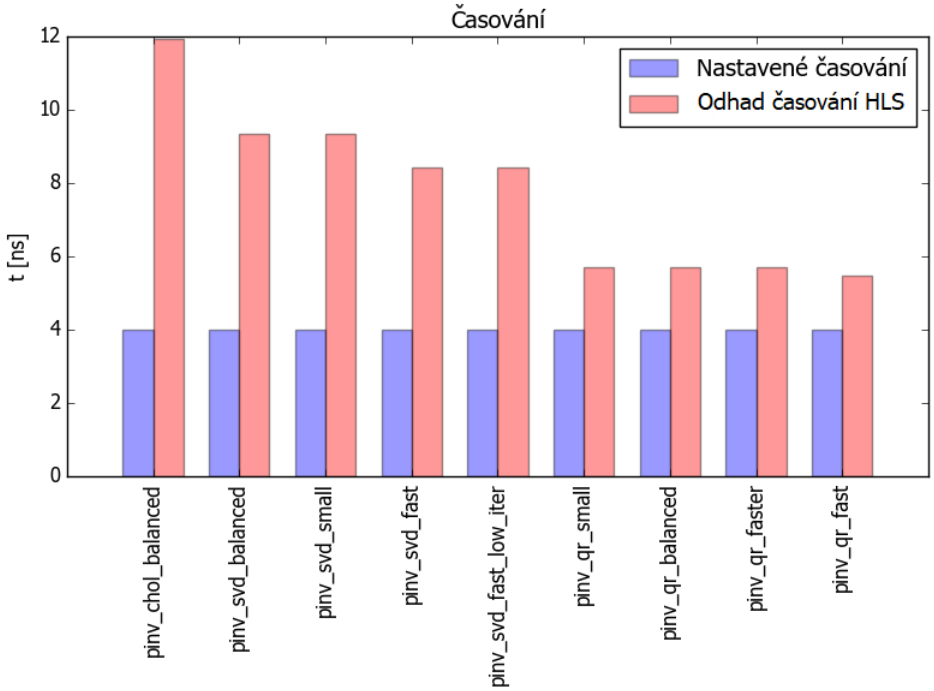
je důležité pro zvolení algoritmu pro konkrétní aplikaci. Výsledné parametry jsou závislé na volbě FPGA obvodu při procesu syntézy. Při srovnání jsou uvažovány varianty pracující s komplexními matice o rozměrech 4×4 a obvod *Zynq 7020*. Následujících graf na obr. 5.14 porovnává výslednou dobu výpočtu v ms získané z výsledků doby výpočtu v počtech hodinových pulsů viz obr. 5.15, tyto výsledky jsou na jednotlivých FPGA obvodech srovnatelné. Poslední graf na obr. 5.16 zobrazuje zvolenou a výslednou periodu hodinových pulsů, která je nejvíce citlivá na zvolené FPGA. Nejdelší doby výpočtu, tedy nejhorších výsledků dosahuje varianta `pinv_svd_balanced` s dobou výpočtu 0,77 ms, na to má vliv především náročný algoritmus singulárního rozkladu. Nejlepšího výsledku dosáhla varianta využívající QR rozklad, konkrétně `pinv_qr_faster` dosahující 0,022 ms.



Obr. 5.14: Srovnání doby výpočtu pseudoinverze pro rozměr matice 4×4



Obr. 5.15: Srovnání doby výpočtu pseudoinverze pro rozměr 4×4



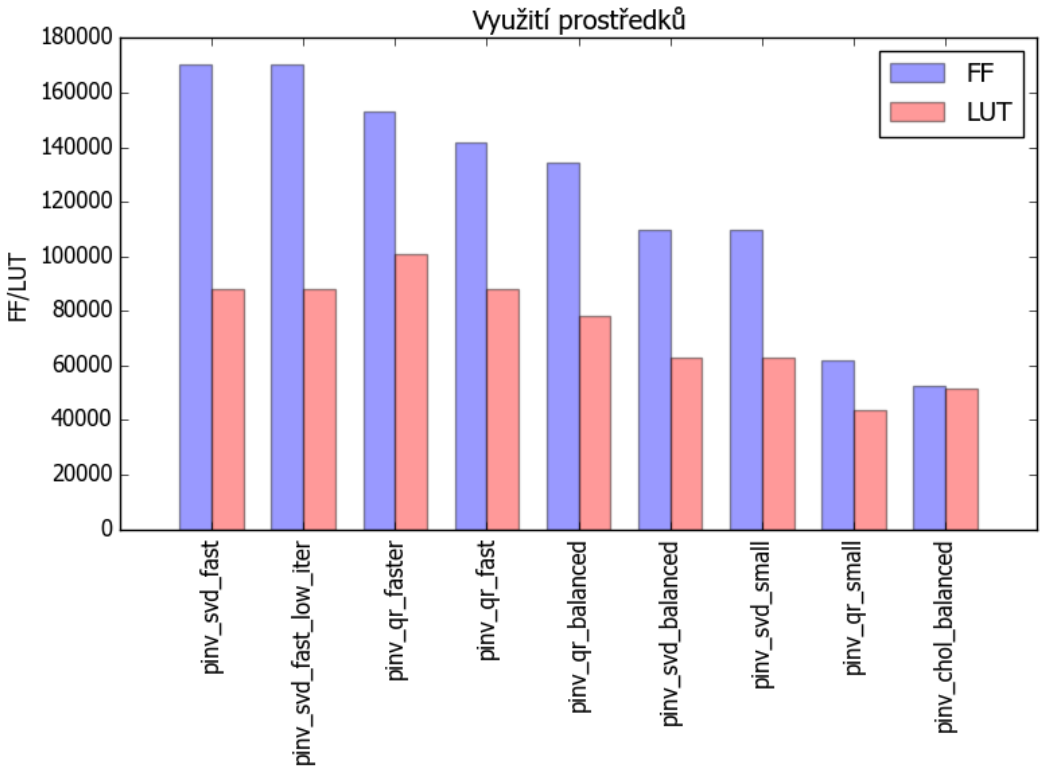
Obr. 5.16: Srovnání časování jednotlivých variant

Varianta	**Doba výpočtu** [-]	**Časování** [ns]	**Doba výpočtu** [ms]
`pinv_chol_balanced`	5116	11,93	0,061
`pinv_svd_balanced`	83292	9,35	0,779
`pinv_svd_small`	50292	9,35	0,470
`pinv_svd_fast`	36424	8,42	0,307
`pinv_svd_fast_low_iter`	22072	8,42	0,186
`pinv_qr_small`	5134	5,68	0,029
`pinv_qr_balanced`	4120	5,68	0,023
`pinv_qr_faster`	3949	5,68	0,022
`pinv_qr_fast`	69080	5,45	0,376

Tab. 5.10: Srovnání variant pro výpočet pseudoinverze komplexní matice 4×4

Srovnání z hlediska využití prostředků

V tabulce 5.11 je shrnuto využití prostředků FPGA, tyto informace jsou získané z reportů vygenerovaných při syntéze ve *Vivado* HLS. Toto srovnání je důležité pro zvolení algoritmu a vhodného FPGA obvodu. Při srovnání jsou uvažovány varianty pracující s komplexními maticemi o rozměrech 4×4. Z následujícího grafu na obr. 5.17 je patrné, že nejvyšší nároky na prostředky FPGA má algoritmus využívající singulárního rozkladu ve variantě `pinv_svd_fast`. Nejméně prostředků využívá varianta `pinv_chol_balanced`, je to zejména proto, že tato varianta jako jediná využívá aritmetiky s pevnou řádovou čárkou, což se projevuje také na numerické přesnosti výpočtu. Zbytek variant využívá prostředky FPGA srovnatelně a podle zvolené optimalizace se pohybují potřebné počty LUT a FF mezi 40 a 80 tisíci.



Obr. 5.17: Využití prostředků výpočtu pseudoinverze pro rozměr 4×4

varianta	FF	LUT	DSP	BRAM
`pinv_svd_fast`	170408	87836	331	44
`pinv_svd_fast_low_iter`	170405	87834	331	44
`pinv_qr_faster`	153341	100813	368	44
`pinv_qr_fast`	141592	87926	296	58
`pinv_qr_balanced`	134450	78360	224	36
`pinv_svd_balanced`	109553	62910	218	44
`pinv_svd_small`	109546	62908	218	44
`pinv_qr_small`	62024	43724	116	34
`pinv_chol_balanced`	52656	51627	54	8

Tab. 5.11: Srovnání jednotlivých variant výpočtu pseudoinverze pro rozměr 4×4

5.4 Ověření bloků

Zde je podrobný popis postupu, kterým je možné exportovat vytvořené HLS bloky pro následný import v Prostřední *Vivado*. Tento postup byl použit pro ověření výpočtů na vývojovém kitu *Z-Turn board*, který obsahuje obvod *Zynq 7020*, zapojený kit

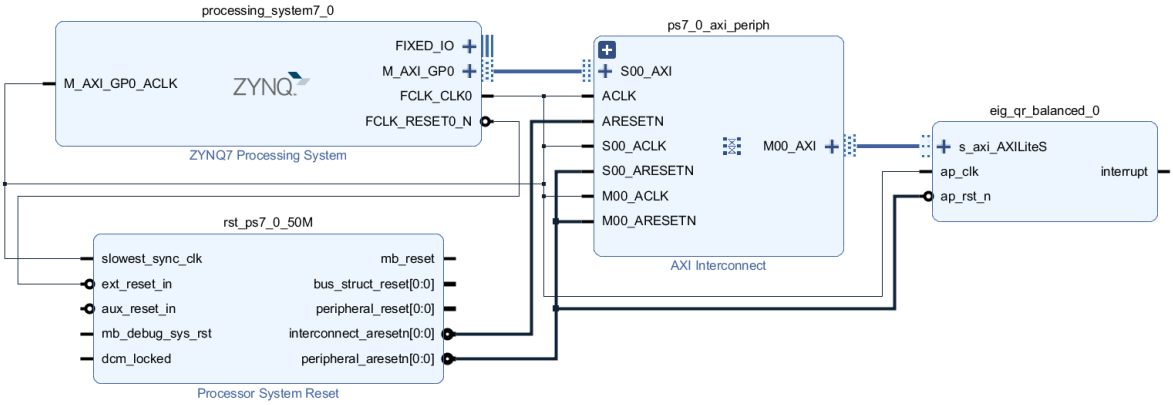
pro ověření je na obr. 5.20. Tento obvod kromě hradlového pole, které je označováno jako PL (*Programable Logic*) a obsahuje Hard IP procesor architektury *ARM Cortex* označovány jako PS (*Procesing System*). PS obsahuje celou řadu periferií, mezi nimi také UART, který je použit pro přenos mezí počítačem a PS.

5.4.1 Export bloků z *Vivado* HLS

Před samotným exportem je potřeba zvolit vhodné rozhraní pro vstupní a výstupní porty. Volba rozhraní se provádí vložením direktiv, v okně *Directive* pravým tlačítkem zvolíme *Insert Directive...* pro vybranou *top* funkci. Ve *Vivado HLS Directive Editor* zvolíme *INTERFACE* a vybereme požadovaný typ ve volbě *Mode*, můžeme zvolit například *s_axilite* nebo jiné rozhraní viz UG902 [11]. Tato volba nám pro zvolenou funkci vytvoří *AXI Lite Slave* rozhraní pro tzv. *handshake* porty (Signály `ap_ready`, `ap_iddle`, `ap_start` atd.) Dále postupujeme stejným způsobem pro jednotlivé argumenty top funkce, ke kterým vytvoříme jednotlivá rozhraní. Díky volbě *AXI Lite Slave* jsou k vygenerovanému IP bloku přiloženy ovladače pro komunikaci s procesorem, který budeme používat, jako prostředníka pro přenos dat mezi ověřovaným blokem a testovacím Programem na PC. Tento postup vychází z informací čerpaných z tutoriálu [12].

5.4.2 Import bloků ve *Vivado*

Vygenerované bloky musíme importovat v prostředí *Vivado* a vložit do *block design*, ve kterém je nakonfigurovaný blok PS *Zynq*. Konfigurace se provádí vložením bloku a nastavením parametrů. Pro otestování je dostačující minimální konfigurace, která se provádí vypnutím periferií kromě *UART* a *AXI lite master* rozhraní. Následně provedeme propojení hodinových signálů a můžeme vložit testovány blok HLS. Více možností pro nastavení vlastností a periferií je popsáno v příručce zabývající se efektivním návrhem na architektuře *Zynq* [14].



Obr. 5.18: Ověření na vývojovém kitu

V tuto chvíli můžeme provést automatické propojení, to spojí příslušné porty bloku PS a PL a dostáváme *block design*, jako je na obr. 5.18. Z *block design* následně vytvoříme HDL *wrapper*, ze kterého vygenerujeme *bitstream*. *Vivado* provede syntézu a implementaci. Zvolíme *export hardware* včetně vložení *bitstreamu* což umožní vytvoření projektu v *Xilinx SDK*, které otevřeme volbou z menu. Dojde k načtení příslušných souboru projektu obsahující obslužné ovladače PS systému a ovladače bloku HLS.

```
void XEig_qr_balanced_Start(XEig_qr_balanced *InstancePtr);
u32 XEig_qr_balanced_IsDone(XEig_qr_balanced *InstancePtr);
u32 XEig_qr_balanced_IsIdle(XEig_qr_balanced *InstancePtr);
u32 XEig_qr_balanced_IsReady(XEig_qr_balanced *InstancePtr);
void XEig_qr_balanced_EnableAutoRestart(XEig_qr_balanced *InstancePtr);
void XEig_qr_balanced_DisableAutoRestart(XEig_qr_balanced *InstancePtr);

u32 XEig_qr_balanced_Get_A_re_BaseAddress(XEig_qr_balanced *InstancePtr);
u32 XEig_qr_balanced_Get_A_re_HighAddress(XEig_qr_balanced *InstancePtr);
u32 XEig_qr_balanced_Get_A_re_TotalBytes(XEig_qr_balanced *InstancePtr);
u32 XEig_qr_balanced_Get_A_re_BitWidth(XEig_qr_balanced *InstancePtr);
u32 XEig_qr_balanced_Get_A_re_Depth(XEig_qr_balanced *InstancePtr);
u32 XEig_qr_balanced_Write_A_re_Words(XEig_qr_balanced *InstancePtr, int offset,
    int *data, int length);
u32 XEig_qr_balanced_Read_A_re_Words(XEig_qr_balanced *InstancePtr, int offset, int
     *data, int length);
u32 XEig_qr_balanced_Write_A_re_Bytes(XEig_qr_balanced *InstancePtr, int offset,
    char *data, int length);
u32 XEig_qr_balanced_Read_A_re_Bytes(XEig_qr_balanced *InstancePtr, int offset,
    char *data, int length);
```

Výpis 5.1: funkce z hlavičkového souboru ovladačů bloku HLS

Zde vytvoříme *Application project*, ve kterém importujeme knihovny a můžeme vytvořit FW pro komunikaci s připojenými HLS bloky.

5.4.3 Testovací *firmware* pro *Zynq* PS

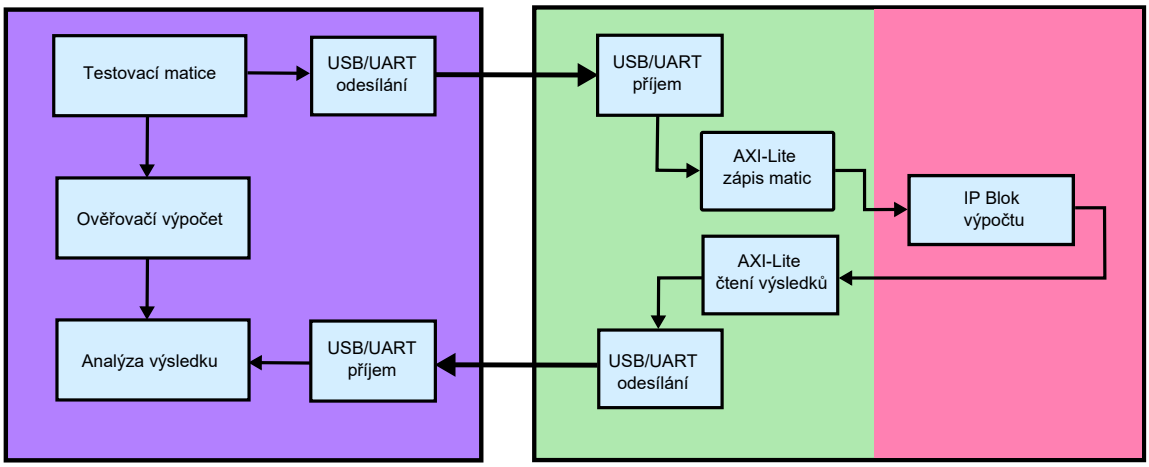
Základem *firmwaru* pro ověření výpočtů jsou funkce ovladačů pro ověřovaný blok a funkce pro čtení a zápis pomocí UART. Po inicializaci těchto periferií je program přiveden do smyčky, ve které čeká na načtení testovacích dat, následuje provedení výpočtu, pomocí funkcí pro výpis proběhne zápis výsledků pro vizuální kontrolu a nakonec přenos samotných výsledků v binární podobě pomocí UART. Proces pokračuje dále ve smyčce načítáním dalších testovacích dat. Pro uchování dat je využit typ *union*, protože funkce pro UART přenos využívají pole `uint8`, funkce pro přenos mezi PL a PS využívají pole typu `uint32` a pří výpisu výsledků pro vizuální kontrolu je potřeba přetypovaní na `float`. Prvky *unionu* umožňují reprezentovat data, jako tyto typy, při zachování stejné adresy v paměti, proto jsou zápisem do jednoho prvku ovlivněny ostatní, reprezentující odlišný datový typ.

Obr. 5.19: Ověření na vývojovém kitu

5.4.4 Testovací skript na PC

V rámci této práce byl v prostředí *Matlab* implementován jednoduchý obslužný skript pro přenos matic a kontrolu výsledků. *Matlab* obsahuje funkce pro přenos po sériové lince, zápis a čtení hodnot probíhá v binární podobě, doplňkově jsou přečteny znaky, které *firmware* poskytuje pro vizuální kontrolu. Skript výsledky po načtení uloží a vypočítá absolutní a relativní odchylky vlastních čísel z FPGA a *Matlab*.

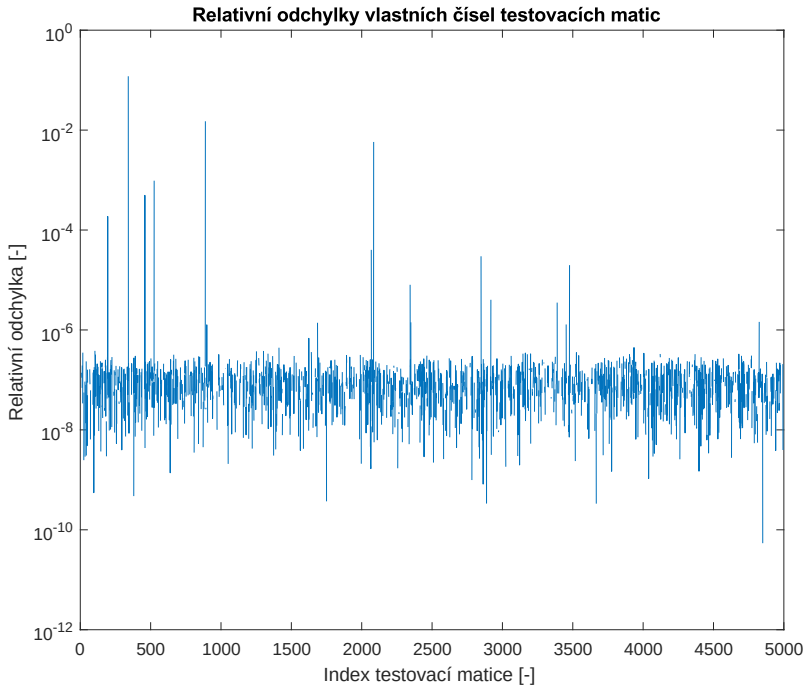
Pro porovnání vlastních vektorů, jsou vypočtené a ověřovací vektory vynásobeny konstantou tak, aby první prvek u obou vektorů byl 1.



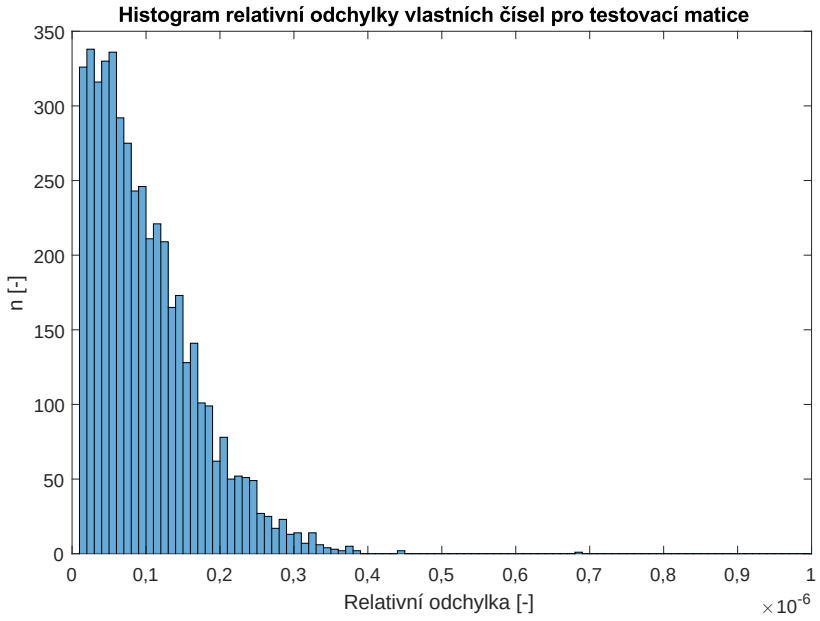
Obr. 5.20: Ověření na vývojovém kitu

5.5 Analýza výsledků s naměřenými daty

Analýza výsledků probíhala způsobem popsaným v předchozí části a testovací matice byly vypočteny na základě dat z měření. Pro ověření byla zvolena varianta `eig_qr_balanced`. Z vypočtených výsledků vlastních čísel byla určena absolutní a relativní odchylka od referenční vlastních čísel vypočítaných pomocí *Matlab* funkce. Průběh relativní odchylky dominantního vlastního čísla je zobrazen v grafu na obr. 5.21, jehož svislá osa je v logaritmickém měřítku. Průměr relativní odchylky je $2{,}845 \cdot 10^{-5}$ a jeho směrodatná odchylka $1{,}7 \cdot 10^{-3}$. Relativní odchylky jsou reprezentovány pomocí histogramu na obr. 5.22, z něhož je patrné, že většina výpočtů proběhla s relativní odchylkou menší než $0{,}3 \cdot 10^{-6}$.



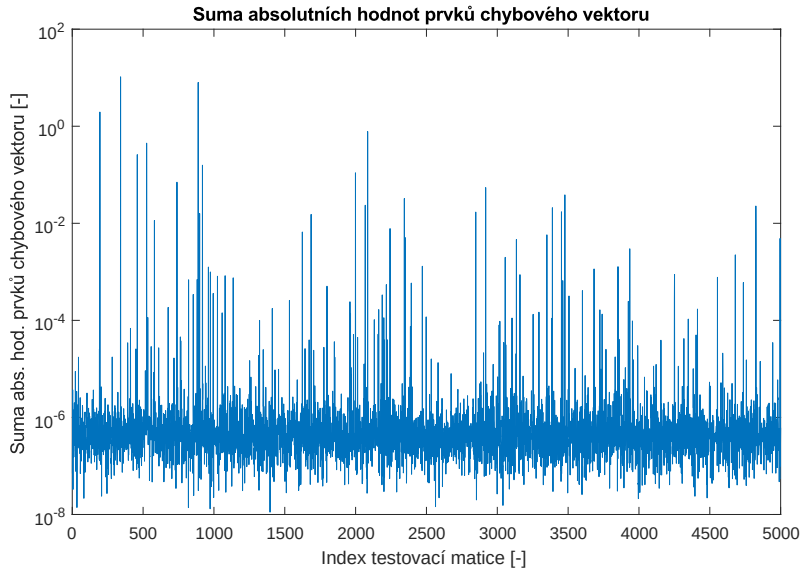
Obr. 5.21: Relativní odchylka dominantního vlastního čísla



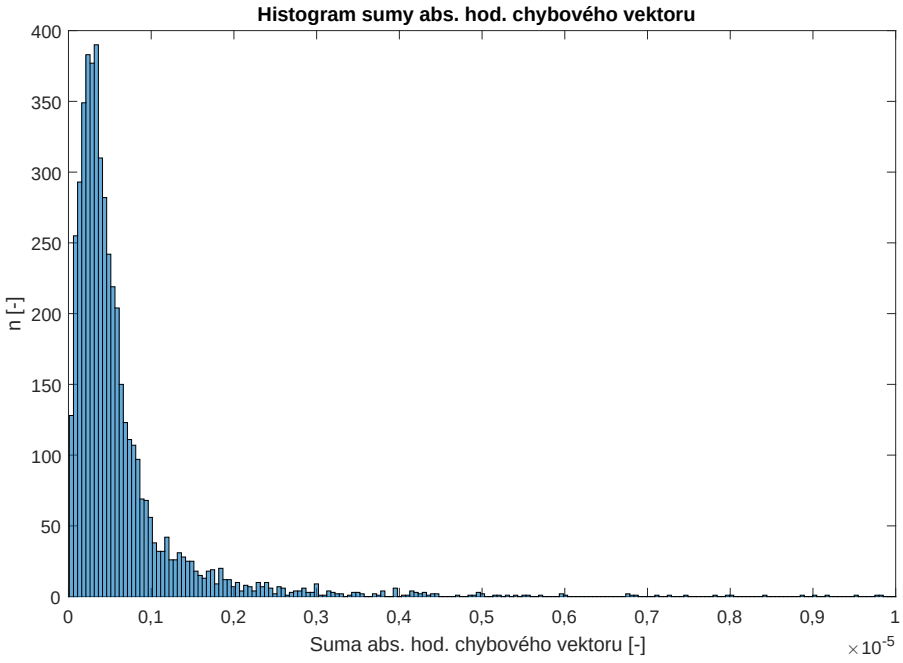
Obr. 5.22: Histogram relativní odchylky dominantního vlastního čísla

Vlastní vektory příslušné dominantnímu vlastnímu číslu byly vynásobeny konstantou tak, aby první prvek vektorů byl 1, následně byl vypočten chybový vektor,

jako rozdíl vypočítaného a referenčního vlastního vektoru. V grafu na obr. 5.23 můžeme vidět součet absolutních hodnot prvků chybového vektoru pro jednotlivé testovací matice. Z průběhu histogramu na obr. 5.24 lze opět vyčíst, že většina vlastních vektorů, měla prvky chybových vektorů pod hranicí $1 \cdot 10^{-6}$.



Obr. 5.23: součet absolutních hodnot prvků chybového vektoru



Obr. 5.24: Histogram součtu absolutních hodnot prvků chybového vektoru

6 ZÁVĚR

Na začátku této práce jsou popsány důležité dekompozice matic, které jsou následně využity v algoritmech pro výpočet vlastních čísel a vektorů a v algoritmech pro výpočet pseudoinverze matice. Velká pozornost byla věnována především QR rozkladu, protože je základem pro nejčastěji využívanou metodu pro výpočet vlastních čísel a vektorů, kterým je QR algoritmus. QR rozklad je možné použít i pro výpočet pseudoinverze pomocí přímého algoritmu nebo prostřednictvím SVD rozkladu, jednou z možností výpočtu SVD rozkladu je opět QR rozklad. Způsobů, jakými se QR rozklad provádí, existuje celá řada. V práci jsou uvedeny a srovnány základní algoritmy, mezi které patří Gram-Schmidtův proces, Householderova reflexe a Givesova rotace. Ze srovnání plyne, že nejlepší vlastnosti má Householderova reflexe, srovnatelného výsledku jsme schopni dosáhnout pomocí modifikovaného Gram-Schmidtova procesu. Givesova rotace vychází ze srovnání náročnosti výpočtu hůře, avšak vyniká možností paralelního zpracování. Z tohoto důvodu je tato varianta použita například i v knihovnách HLS. Příbuzný základ má i Jacobiho metoda výpočtu vlastních čísel a vektorů, která se dá využít pro symetrické matice, případně Jacobiho metoda pro výpočet singulárního rozkladu.

Pro implementaci algoritmů na obvodu FPGA byl použit nástroj *Vivado* HLS. Po obecném úvodu do problematiky FPGA obvodů a jejich programování byly detailně popsány možnosti nástrojů *Vivado* HLS. Byla zmíněna struktura projektu, vstupy a výstupy, které HLS poskytuje, přehled datových typů a důležitých funkcí z knihovny pro lineární algebru, které byly následně využity pro implementaci výpočtů. V této práci je také uveden popis možností, kterými lze ovlivnit výstupní RTL a optimalizovat tak dobu výpočtu a využití prostředků FPGA obvodu. Tyto postupy byly využity při implementaci výpočtů, analýze a ověřování výsledných parametrů. Pomocí *Vivado* HLS byl implementován výpočet vlastních vektorů pomocí QR Algoritmu a singulárního rozkladu. Stejné dekompozice byly využity pro implementaci výpočtu pseudoinverze, pro další varianty tohoto algoritmu byla využita Choleského dekompozice s možností využití formátu s pevnou desetinnou čárkou. Pro jednotlivé metody byly dále vytvořeny varianty lišící se volbou optimalizačních parametrů. Jednotlivé implementované varianty byly srovnány z hlediska využití prostředků FPGA a parametrů časování. Ze srovnání mají velmi dobré výsledky varianty `eig_qr_fast` a `eig_qr_balanced`.

Zvolený blok `eig_qr_balanced` byl ověřen na vývojovém kitu *Z-Turn Board*, pomocí vhodného nastavení rozhraní pro vstupní a výstupní porty bylo vytvořeno IP jádro s rozhraním *AXI-Lite*, které umožňuje snadné spojení s PS částí obvodu *Zynq*. Pomocí ARM procesoru a periferie UART byla přenášena testovací data mezi IP jádrem a softwarem pro testování. Výsledky výpočtů byly srovnány s referenčními

výsledky pomocí *Matlab*. Pro vlastní čísla byla určena absolutní a relativní odchylka, která se pro dominantní vlastní číslo pohybovala v řádu 10^{-7} až 10^{-6}. Výsledky vektorů byly nejprve sjednoceny tak, aby první prvek byl 1 a ze zbývajících prvků byla určena odchylka, která se pohybovala řádově 10^{-6} až 10^{-5}.

Výsledkem této práce jsou IP bloky pro zadané výpočty, parametry implementovaných funkcí je možné měnit na základě požadavků plynoucích z vlastností celého systém použít implementované bloky nebo upravit hlavičkové a definovat nové varianty. V budoucnu mohou IP bloky složit jako základ pro v algoritmy v dohledových systémech letového provozu, které využívají multikanálový příjem a na základě fázových poměrů na jednotlivých anténách určují směr příchodu případně separují smíšené signály, což umožní jejich dekódování.

LITERATURA

[1] GOLUB, G. H.; VAN LOAN, C. F. *Matrix Computations* 3. vyd. Baltimore: Johns Hopkins University Press, 1996. ISBN 0-8018-5413-X.

[2] MEYER, C. D. *Matrix analysis and applied linear algebra.* Philadelphia: Society for Industrial and Applied Mathematics, c2000. ISBN 978-0898714548.

[3] PERSSON, P. *Lecture 5 Gram-Schmidt Orthogonalization.* Introduction to Numerical Methods. MIT 18.335J / 6.337J September 21, 2006. Dostupné z URL: <https://www.mdh.se/polopoly_fs/1.32780!/Menu/general/column-content/attachment/MAA704_lecture_ce_9_bbversion%20(2).pdf>.

[4] ENGSTROM, C. *Numerical methods for computing eigenvalues and eigenvectors.* MIT Lecture-notes December 14, 2012. Dostupné z URL: <https://ocw.mit.edu/courses/mathematics/18-335j-introduction-to-numerical-methods-fall-2010/lecture-notes/MIT18_335JF10_lec10a_hand.pdf>.

[5] TICHÝ, P. *3.Ortogonální transformace a QR rozklady.* 10.října 2012. Dostupné z URL: <http://home.zcu.cz/~ptichy/kma/download/texts/NA_03.pdf>.

[6] JIA, Y. *Singular Value Decomposition.* (Com S 477/577 Notes) Sep 12, 2017. Dostupné z URL: <http://web.cs.iastate.edu/~cs577/handouts/svd.pdf>

[7] CLINE, A. K.; DHILLON, S. *Computation of the Singular Value Decomposition.* The University of Texas at Austin. <http://www.cs.utexas.edu/users/inderjit/public_papers/HLA_SVD.pdf>.

[8] ANDERSON, P. J.; LOIZOU, G. *A Jacobi Type Method for Complex Symmetric Matrices.* Handbook Series Linear Algebra. Received May 17, 1974.

[9] COURRIEU, P. *Fast Computation of Moore-Penrose Inverse Matrices* Neural Information Processing - Letters and Reviews Vol.8, No.2, August 2005 <https://arxiv.org/ftp/arxiv/papers/0804/0804.4809.pdf>

[10] COMON, P.; JUTTEN, C. *Handbook of Blind Source Separation.* Independent Component Analysis and Applications. October 6, 2009.

[11] *High-Level Synthesis*, Vivado Design Suite User Guide. Xilinx, UG902 (v2017.1) April 5, 2017.

[12] *Vivado Design Suite Tutorial*, High-Level Synthesis, UG871 (v2017.1) May 5, 2017

[13] *UltraFast High-Level Productivity Design Methodology Guide*, Xilinx, UG1197 (v2018.1) April 4, 2018

[14] *Zynq-7000 All Programmable SoC: Embedded Design Tutorial,* A Hands-On Guide to Effective Embedded System Design UG1165 (v2015.1) April 23, 2015

[15] *7 Series FPGAs Configurable Logic Block,* User Guide, UG474 (v1.8) September 27, 2016

[16] Vivado Design Suite User Guide, *Implementation,* UG904 (v2018.1) April 4, 2018

[17] Vivado Design Suite User Guide, *Synthesis,* UG901 (v2018.1) April 13, 2018

[18] SPONSOR a MICROPROCESSOR STANDARDS COMMITTEE OF THE IEEE COMPUTER SOCIETY. *IEEE standard for floating-point arithmetic.* New York, NY: Institute of Electrical and Electronics Engineers, 2008. ISBN 9780738157528.

[19] *7 Series FPGAs SelectIO Resources* User Guide UG471 (v1.10) May 8, 2018

[20] *7 Series FPGAs GTX/GTH Transceivers* User Guide UG476 (v1.12) December 19, 2016

[21] *Reduce Power and Cost by Converting from Floating Point to Fixed Point* White Paper: Floating vs Fixed Point Xilinx WP491 (v1.0) March 30, 2017

SEZNAM SYMBOLŮ, VELIČIN A ZKRATEK

HLS	Nástroje Xilinx Vivado High-Level Synthesis
SVD	Singular-value decomposition – Singulární rozklad
RTL	Register-transfer-level – Reprezentace na úrovni logických hradel
FPGA	Field Programmable Gate Array – Programovatelná hradlová pole
HDL	Hardware description language – Jazyk pro popis číslicových obvodů
VHDL	VHSIC Hardware Description Language - HDL Jazyk
VHSIC	Very High Speed Integrated Circuit
ASIC	application-specific integrated circuit
FFT	Fast Fourier transform
DSP	digital signal procesing – číslicové zpracování signálu
MAC	multiply and acummulate
IP	Intellectual Property – Označení PFGA subsystémů
AXI	Advanced eXtensible Interface – vysokorychlostní komunikační rozhraní
FSM	Finite-state machine – Konečný automat
FF	Flip-flop – Logická buňka v FPGA
LUT	Lookup table – Vyhledávací tabulka (paměťová buňka v FPGA)
DSP48	Výpočetní jednotka v FPGA
BRAM	Block RAM
FFT	Fast Fourier transform – Rychlá Fourierova transformace
FIR	Finite impulse response – Filtr s konečnou impulzní odezvou
HW	Hardware
ALU	Arithmetic logic unit– Aritmeticko-logická jednotka
II	Initial Interval
RAM	Random-access memory – typ paměti
MIMO	Multiple-input multiple-output – Systém v více vstupy a výstupy

SEZNAM PŘÍLOH

A ZDROJOVÉ KÓDY

Zde jsou umístěny výpisy zdrojových kódů, které vznikly v rámci diplomové práce. Nejedná se o veškeré zdrojové kódy, ale pouze o důležité algoritmy nebo ukázky některých principů.

A.1 Zdrojové kódy pro Matlab

A.1.1 Jacobiho metoda

Klasická Jacobiho metoda

```
function [V,D] = jacobi1complex(A,epsilon,show)
%JACOBI1   The original Jacobi's method of iteration is used
%          to compute the eigenpairs of a symmetric matrix.
% Sample call
%   [V,D] = jacobi1(A,epsilon,show)
% Inputs
%   A          matrix
%   epsilon    convergence tolerance
% Return
%   V          solution: matrix of eigenvectors
%   D          solution: diagonal matrix of eigenvalues
% Upraveno pro komplexní čísla na základě:
% NUMERICAL METHODS: MATLAB Programs, (c) John H. Mathews 1995
% Algorithm 11.3 (Jacobi Iteration for Eigenvalues and Eigenvectors
   ).
% Section 11.3, Jacobi's Method, Page 571


if nargin==2, show = 0; end
D = A;
[n,n] = size(A);
V = eye(n);
done = 0;
working = 1;
stat = working;
cntr = 0;
[m1 p] = max(abs(D-diag(diag(D)))); % Select element
[m2 q] = max(m1);                   % element of largest
p = p(q);                           % magnitude.
while (stat==working),
  [p q]
  fi1 = atan(imag(D(p,q))/real(D(p,q)));
  fi2 = atan(2*abs(D(p,q))/(D(p,p)-D(q,q)));
```

```
33    teta1 = (2*fi1 - pi)/4;
34    teta2 = fi2/2;
35    Rpp = -i*exp(-i*teta1)*sin(teta2);
36    Rpq = -exp(i*teta1)*cos(teta2);
37    Rqp = exp(-i*teta1)*cos(teta2);
38    Rqq = i*exp(i*teta1)*sin(teta2);
39    R = [Rpp Rpq; Rqp Rqq];
40    D([p q],:) = R'*D([p q],:);
41    D(:,[p q]) = D(:,[p q])*R;
42    V(:,[p q]) = V(:,[p q])*R;
43    cntr = cntr+1;
44    if show==1,
45      home; if cntr==1, clc; end;
46      disp(['Jacobi iteration No. ',int2str(cntr)]),disp(''),...
47      disp(['Zeroed out the element  D(',num2str(p),',',num2str(q),')
          = ']),...
48       disp(D(p,q)),disp('New transformed matrix  D = '),disp(D)
49    end
50    [m1 p] = max(abs(D-diag(diag(D))));
51    [m2 q] = max(m1);
52    p = p(q);
53    if cntr>50, stat = done; end
54  end
55  D = diag(diag(D));
```

Výpis A.1: Funkce pro výpočet vlastních čísel a vektorů Jacobiho metodou

Cyklická Jacobiho metoda

```
1  function [V,D] = jacobi2complex(A,epsilon,show,max_iter)
2  %JACOBI2   The cyclic Jacobi's method of iteration is used
3  %          to compute the eigenpairs of a symmetric matrix.
4  % Sample call
5  %   [V,D] = jacobi2(A,epsilon,show)
6  % Inputs
7  %   A          matrix
8  %   epsilon    convergence tolerance
9  % Return
10 %   V          solution: matrix of eigenvectors
11 %   D          solution: diagonal matrix of eigenvalues
12 % Upraveno pro komplexní čísla na základě:
13 % NUMERICAL METHODS: MATLAB Programs, (c) John H. Mathews 1995
14 % Algorithm 11.3 (Jacobi Iteration for Eigenvalues and Eigenvectors
     ).
15 % Section 11.3, Jacobi's Method, Page 571
16 if nargin==2, show = 0; end
```

```
D = A;
[n,n] = size(A);
V = eye(n);
cnts = 0;
cntr = 0;
done = 0;
working = 1;
stat = working;
for indx= 1:max_iter,
   cnts = cnts+1;
   t = sum(diag(D));
   stat = done;
    for q = 1:(n-1),
      for p = (q+1):n,
          vnitrek(p, q)
          vnitrek(q, p)
            cntr = cntr+1;
          if show==1,
            home; if cntr==1, clc; end;
            disp(['Jacobi iteration No. ',int2str(cntr)]),disp('')
                ,...
            disp(['Zeroed out the element  D(',num2str(p),...
                  ',',num2str(q),') = ']),disp(D(p,q)),...
             disp('New transformed matrix  D = '),disp(D)
          end
      end
    end
end
D = diag(diag(D));
function [] = vnitrek(p,q)
fi1 = atan(imag(D(p,q))/real(D(p,q)));
          fi2 = atan(2*abs(D(p,q))/(D(p,p)-D(q,q)));
          teta1 = (2*fi1 - pi)/4;
          teta2 = fi2/2;
          Rpp = -1i*exp(-1i*teta1)*sin(teta2);
          Rpq = -exp(1i*teta1)*cos(teta2);
          Rqp = exp(-1i*teta1)*cos(teta2);
          Rqq = 1i*exp(1i*teta1)*sin(teta2);
          R = [Rpp Rpq; Rqp Rqq];
          D([p q],:) = R'*D([p q],:);
          D(:,[p q]) = D(:,[p q])*R;
          V(:,[p q]) = V(:,[p q])*R;
end
end
```

Výpis A.2: Cyklická varianta Jacobiho metody

A.1.2 Ortogonalizace

modifikovaný Gram–Schmidtův proces

```
for j = 1:n
    v = A(:,j);
    for i =1:j-1
        R(i,j) = Q(:,i)'*v;
        v = v -R(i,j)*Q(:,i);
    end
    R(j,j)=norm(v);
    Q(:,j) =v/R(j,j);
end
```

Výpis A.3: Modifikovaný Gram–Schmidtův ortogonalizační proces

A.1.3 Metoda Rayleigho podílu

```
function [sigma,x,iter] = rayqot2(A,sigma0,ep,numitr)
%RAYQOT Rayleigh quotient iteration
%[sigma,x,iter] = rayqot(A,x0,ep,numitr) computes an approximate
%eigenpair(sigma,x) of a matrix A using Rayleigh-Quotient iteration
    .
%x0 is the initial approximation to the eigen vector x.  ep is the
    tolerance
%numitr is the user-supplied number of iteration.  On output, if
    the
%Rayleigh quotient iteration converged, iter contains the iteration
     number
%needed to converge. If the iteration did not converge, iter
    contains
%the value of numitr.
%This program implements Algorithm 8.5.3 of the book.
%input  : Matrix A, vector x0, scalar ep and integer numitr
%output : Scalar sigma, vector x and integer iter

  [m,n] = size(A);
        if m~=n
          disp('matrix A  is not square')  ;
          return;
        end;
        prevsig = 0;
        xhat = (A-sigma0 * eye(n,n))\ones(n);
         x = xhat/max(xhat);
        for k = 0 :  numitr
         iter = k;
```

```
24          sigma = (x'*A*x)/(x'*x);A
25          xhat = (A-sigma * eye(n,n))\x;
26          x = xhat/max(xhat);
27          if  norm( (A-sigma * eye(n,n))*x )  < ep
28             return;
29          end
30          prevsig = sigma;
31        end;
32  end
```

Výpis A.4: Metoda Rayleigho podílů

A.1.4 Generování testovacích dat

```
1  function [ A ] = genMat( N )
2  % genMat generuje komplexni pozitivne definitni matici
3  % % N rozmer matice
4  v = complex(randn(N,1),randn(N,1));
5  A = v * v';
6  % Pridani kladne realne konstanty k diagonale = pridani sumu
7  sigma = sqrt(norm(v))/5; % 5 x mensi
8  A = A + sigma^2 * eye(N,N);
9  % Normalizace na hodnotu pro zajisteni vlastnich cisel a vlastnich
       vektoru
10 % v rozsahu dvojkoveho doplnku
11 Amax = max(max(A));
12 A = A/Amax()/N;
13 end
```

Výpis A.5: Funkce pro generování testovací matice

```
1  %clc
2  clear all
3
4      %A = genMat(4);
5      A = complex(randn(4,4),randn(4,4))
6      A= A*A'
7      A = -A
8      [eig_vec eig_val ] = eig(A);
9      [ U E V] = svd(A);
10     e= diag(E)
11
12     d = diag(eig_val)
13     dd =d.*conj(d)
14     sqrt(dd)
```

Výpis A.6: Skript pro generování testovacích dat

A.2 Zdrojové kódy pro HLS

A.2.1 Příklad struktury *test bench*

```
#include "hier_func.h"
int main() {
  // Data storage
  int a[NUM_TRANS], b[NUM_TRANS];
  int c_expected[NUM_TRANS], d_expected[NUM_TRANS];
  int c[NUM_TRANS], d[NUM_TRANS];
  //Function data (to/from function)
  int a_actual, b_actual;
  int c_actual, d_actual;
  // Misc
  int    retval=0, i, i_trans, tmp;
  FILE *fp;
  // Load input data from files
  fp=fopen(tb_data/inA.dat,r);
  for (i=0; i<NUM_TRANS; i++){
    %fscanf(fp, %d, &tmp);
    a[i] = tmp;
  }
  fclose(fp);
  fp=fopen(tb_data/inB.dat,r);
  for (i=0; i<NUM_TRANS; i++){
    fscanf(fp, %d, &tmp);
    b[i] = tmp;
  }
  fclose(fp);
  // Execute the function multiple times (multiple transactions)
  for(i_trans=0; i_trans<NUM_TRANS-1; i_trans++){
    //Apply next data values
    a_actual = a[i_trans];
    b_actual = b[i_trans];
    hier_func(a_actual, b_actual, &c_actual, &d_actual);
    //Store outputs
    c[i_trans] = c_actual;
    d[i_trans] = d_actual;
  }
  // Load expected output data from files
  fp=fopen(tb_data/outC.golden.dat,r);
  for (i=0; i<NUM_TRANS; i++){
    fscanf(fp, %d, &tmp);
    c_expected[i] = tmp;
  }
  fclose(fp);
  fp=fopen(tb_data/outD.golden.dat,r);
```

```
44    for (i=0; i<NUM_TRANS; i++){
45      fscanf(fp, %d, &tmp);
46      d_expected[i] = tmp;
47    }
48    fclose(fp);
49  // Check outputs against expected
50    for (i = 0; i < NUM_TRANS-1; ++i) {
51      if(c[i] != c_expected[i]){
52        retval = 1;
53      }
54      if(d[i] != d_expected[i]){
55        retval = 1;
56      }
57    }
58    // Print Results
59    if(retval == 0){
60      printf(    *** *** *** *** \n);
61      printf(    Results are good \n);
62      printf(    *** *** *** *** \n);
63    } else {
64      printf(    *** *** *** *** \n);
65      printf(    Mismatch: retval=%d \n, retval);
66      printf(    *** *** *** *** \n);
67    }
68  // Return 0 if outputs are corre
69    return retval;
70  }
```

Výpis A.7: Příklad struktury *test bench*

B ELEKTRONICKÉ PŘÍLOHY